Carnegie Mellon
Software Engineering Institute

Construction and Deployment Scripts for COTS-Based, Open Source Systems

Wilfred J. Hansen
January 2000

TECHNICAL REPORT
CMU/SEI-99-TR-013
ESC-TR-99-013

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Construction and Deployment Scripts for COTS-Based, Open Source Systems

CMU/SEI-99-TR-013
ESC-TR-99-013

Wilfred J. Hansen

January 2000

COTS-Based Systems Initiative

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Norton L Compton

Norton L. Compton, Lt Col., USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

List of Tables

Abstract

Construction/deployment scripts direct the compilation of sources to executables and the installation of those executables. This report details the construction/deployment scripts developed at the Software Engineering Institute for the GEE project. GEE, a Generic Enterprise Ensemble, is a prototypical three-tier information system incorporating a number of commercial off-the-shelf (COTS) products. The scripts for GEE were challenging because we wanted a self-contained package of scripts and source files from which the system could be built and deployed either by us at our site or by customers at their sites. The COTS products we used—Java, an Oracle database, the Visibroker implementation of CORBA (the Common Object Request Broker Architecture), and the Netscape Web browser and server—added challenges in installation, version change, process initiation, and communication rendezvous. This report describes the challenges and how our solutions exploited the principles of "Repeat not" and "Delay binding." Lessons learned are reported elsewhere.[1]

[1] Hansen, W. J. "Deployment Descriptions in a World of COTS and Open Source," Ninth International Symposium on System Configuration Management (SCM-9). Toulouse, France, Sept. 1999. Heidelberg: Springer Verlag LNCS, 1999. Related source files are available on the WWW <URL: http://www.sei.cmu.edu/staff/wjh/geebuild.tar>.

1 Introduction

Before productive use, a software system must be written as a set of source files, built from these sources, tested, corrected, and finally installed into global directories for access by users. Together, these steps constitute the "construction and deployment" of the system. Each step can be initiated manually with a sequence of commands: run a compiler, run a linker, start a server process, start a client process, copy files from one place to another, and so on. However, as the size of the system increases, so do the number and complexity of these commands. The complexity is further compounded as the system is re-targeted to additional hardware and operating system platforms. It may still be possible for a developer to enter the commands manually, but it is virtually impossible to document these commands sufficiently so that their sequence can be repeated by anyone else. Instead, the commands must be encapsulated in some sort of script file. Indeed, developers routinely simplify and accelerate development by use of such files, which we can call *construction/deployment scripts*.

Construction of a system as a monolithic whole with all sources under control of the developers was once the norm, but is no longer. Today commercial off-the-shelf (COTS) products are organic constituents of systems; they provide databases, interprocess communication, user interfaces, Web tools, and many other facilities. Incorporating these products into construction/deployment scripts poses a number of new challenges. For instance, the work reported here revealed problems with installing COTS products, accommodating their version changes, starting their processes, and establishing communication for each with the rest of the system.

This report describes the nature and details of the construction/deployment scripts developed at the Software Engineering Institute for the GEE (Generic Enterprise Ensemble) project [Wallnau 98]. GEE is an enterprise-level information system developed as an ensemble of COTS products and locally written code. The system was planned for distribution in open-source form so that developers might study GEE to learn how its collection of COTS products can be used together. Moreover, some generic portions of GEE are designed for direct incorporation into enterprise systems. In a sense, GEE provides a sort of fill-in-the-blanks enterprise information system; we call it a "genotype" because, in addition to the functions of a prototype, it also serves as a framework for building a real-life system. COTS products used in GEE include Java [Sun 99a], Netscape servers and browsers [Netscape 99], the Visibroker CORBA tool [Inprise 99], an Oracle database [Oracle 99], and Live Software's JRun tool for Java servlets [Live Software 99].

Some lessons learned from this work have been reported in a technical paper [Hansen 99]. The scripts described here are available on the Web [Hansen 98b].

1.1 Principles

Much of the design of the GEE construction scripts is based on this first principle:

Repeat not.

In other words, do not put the same information in two places. This is not laziness; carrying out this plan in practice is often considerably more work. The point is that build commands change frequently, especially while creating them, and isolating each aspect to a single point of change simplifies modification. The key to avoiding repetition is to give a name to an entity, define the name once, and then use the name everywhere else to refer to the entity.

Of course, this first principle is common in computing. It is the basis for the notion of subroutines and functions. It is the foundation of the notion of modularity and its modern incarnation, object-oriented programming. In each case, the fundamental idea is to define a name and use the name in place of some value that might change.

Corollary 1. Move commonalties to a central file. It is not enough that nothing be repeated. It is also necessary that the definition of an entity be easy to find so that the definition can be revised. The best way to do this is to collect definitions in one central location; not only does this aid revision, but it also makes it possible to ensure that

- the proper definition will be accessed for each appearance of the name
- when one change is made, the appropriate ancillary changes are also made
- site-dependent information can be collected in one small area
- all changes can be made in a few files rather than making changes to a plethora of files scattered across the system sources

Avoiding scattered changes reduces the intellectual effort of reading each file and the attendant risk of introducing errors while making the change.

Corollary 2. Site-dependent information should never appear in more than one place. Site-dependent information is information that is likely to change when the system is moved to another site. It is bad enough that installation at a new site requires changing any file. The need to change multiple files is a sure recipe for problems.

Related to the first principle is a second:

Defer binding.

This principle, too, has a long history in software. For command scripts, the meaning is to avoid inserting into a file some constant where there should be a variable because the information may change. Avoid constants for pathnames, database names, URLs, and so on. Any such constants reduce the flexibility of the software; for instance, pathnames make it impossible to install the software in a different directory location without rebuilding it.

1.2 Tasks of the GEE Construction Scripts

To support both local development and eventual distribution, the construction/deployment scripts had to satisfy several needs:

- Independent testing. Allow each developer to test a newly revised component without installing it into a full-blown version of the system.
- Partial build. Permit independent work on separate system components.
- Developer full system build as an option. This is used for testing fundamental revisions.
- Full build and install. This is a customer installation option and also permits a nightly development system rebuild to ensure that the released version is current.
- Multiple versions. Old versions must remain buildable and installed in order to, for example, check that functionality has not been lost in a revision.
- Multiple platforms. Support building on different hardware and operating systems.
- Build at a foreign site. Enable a distribution recipient to build and install the system.
- Extension. Enable developers at a site to augment GEE with additional components.

The first four requirements require distinct build scenarios, as diagramed in Figure 1, while the last four are met by other provisions. The Source Archive contains all source files that contribute to the build, including the scripts. In each of the four scenarios, sources are copied from the archive and compiled into object files. They may be tested as shown by dotted lines from objects to executions or they may be installed into global directories and executed from there. In the subset build scenario, the subset objects are augmented with those from an installation directory to form a complete executable.

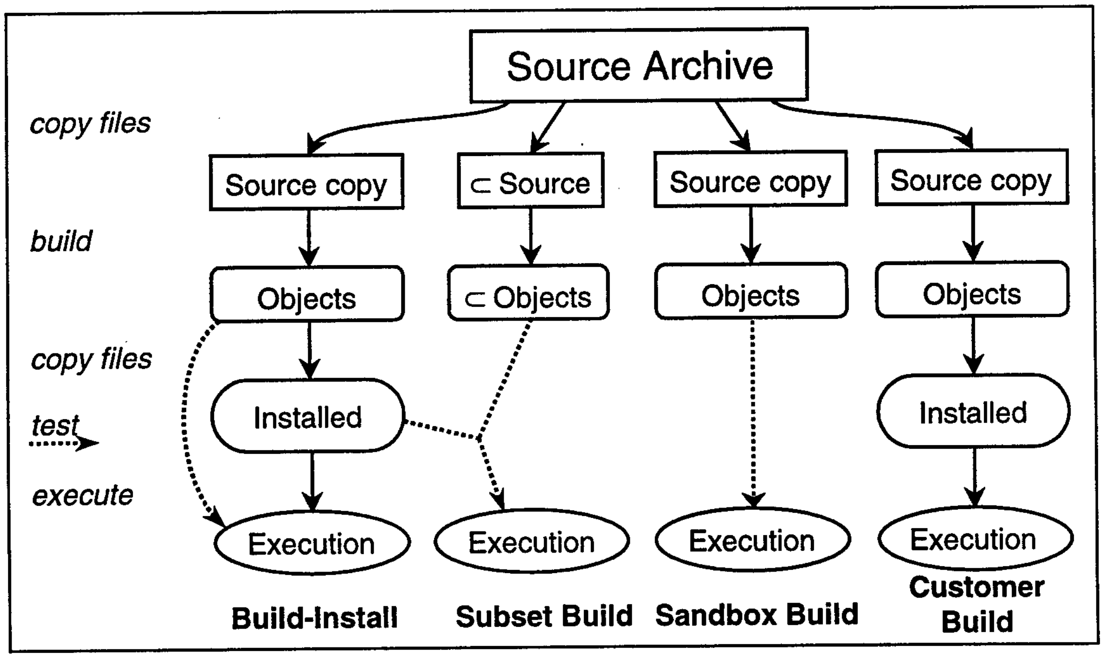


Figure 1: Various Build Scenarios

The customer build scenario on the far right of Figure 1 is the basic process. Both source and object files are in directories private to the developer or installer. After the build, selected object files are installed into system directories, from which general users execute them. For delivery to most user sites, the objects are transmitted to the site and installed in system directories there.

The build-install scenario at the left is similar to the customer build scenario, but here the dotted line shows the installer's option to test the object files by initiating an execution directly, without installation. At a site where a full build-install has been done, an individual developer may change a system subset or "component" with a subset build. In this scenario, a subset of the sources is copied to a subset source directory and then compiled/linked to produce a subset of the object files. These are then combined with other objects from the installed version to perform a test execution. This same scenario can be used by a developer at a customer site to develop new components satisfying local requirements. Other developers with more extensive or sensitive changes may do a full sandbox build. In this scenario, the full system is built and tested directly from the object files.[2]

Some recent research has centered on the installation or "deployment" task and has developed specific tools in that arena [Hall 97, Hansen 99]. For GEE, it seemed appropriate to continue the traditional practice of doing installation with the same construction script tools that are used for system build. Not only does this approach ensure that the build is made to be up-to-date before any install; it also allowed us to exploit earlier experience since the GEE script mechanism is loosely based on that of Andrew [Andrew 96], an integrated word processing and document object system developed at Carnegie Mellon University.

The next chapter describes infrastructure tools used for the GEE source archive and construction. Construction is more specifically described in the third chapter, and later chapters describe the problems encountered with various COTS products. Since there are many problems and solutions, the text in this report uses a special form as in

Problem: How can we support the many build scenarios?

Solution: Write construction/deployment scripts in *imake* and *make*, as shown below.

Italics in text distinguish *command names* or *file names*, where the latter usually have a dot in them. Directory names are file names with a trailing slash. A sans-serif font has been used for sample code that would appear in a program or script. Serif text within sample code is a comment that describes omitted code.

[2] Only in exceedingly rare circumstances should installation be done from a subset or sandbox build; attempts to do so have exhibited a considerable probability of disrupting normal usage.

2 Concepts and External Tools

The GEE implementations of the build scenarios of Figure 1 are based on a few pre-existing tools: *rcs*, *imake*, and *make*. The *rcs* tools—augmented for GEE—provide for exchanging files between the Source Archive and a "build tree" of directories for building from the source. *Imake* builds a construction/deployment script in a build directory, and *make* is then called on to execute the script.

2.1 Build Tree, Source Archive, and *rcs*

A traditional file organization for building from a set of sources is to devote one directory to each type of file (Java sources in one, shell scripts in another, and so on). This organization simplifies the construction scripts for each directory, since every file is treated the same. However, when a functional change is made to the system, it will often involve files of all types, thus requiring changes to every build directory. To minimize this sort of disruption, and for other reasons, we chose a different structure for GEE's sources.

For building GEE, the sources are arranged in a hierarchical tree of directories, each containing the source for one "component." A component is a logical subset of the system chosen at a size that is, more or less, what a developer will create in a week. One objective in defining a component is that it be separable from the rest of the system; ideally, developers working on a single component can have confidence that changes to it will not affect or be affected by changes to other components. Similarly, advanced users should be able to develop system adjuncts as components without having to be concerned with sources in other directories. As an example of component nesting, consider the build tree in Figure 2. Here, the top-level *GEEBuildTree/* directory, with the abbreviated name *BT,* has the outermost component, *src,* as a subdirectory. *Src/* has two components—*config/* and *compx/*—the latter of which also has two components. Each directory has an *Imakefile* and other representative files.[3]

[3] In UNIX, file and directory names are arbitrary strings that do not contain slashes, so it is only by convention that names end with an "extension" of a dot and a few letters. By convention, the names of Java source files end in *.java,* and the result of a compilation is a file with the same name but the new extension *.class*. A UNIX directory name is separated from a subordinate file or directory name with a slash. The top-level, or *root*, directory has the empty string as its name, so absolute pathnames begin with */*.

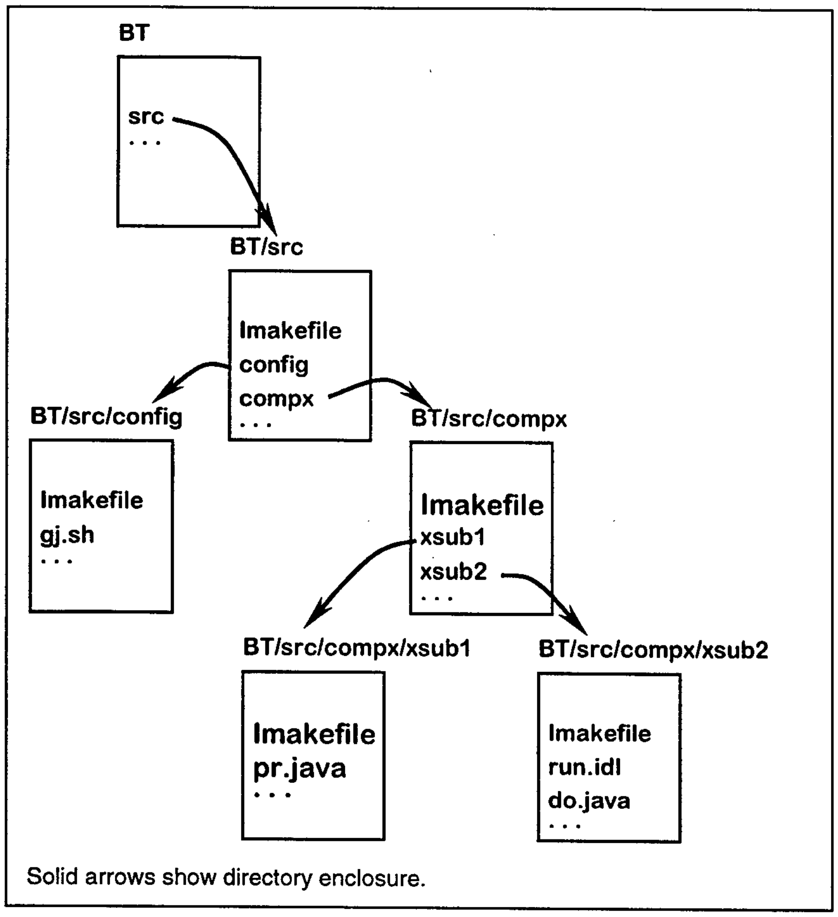


Figure 2: Tree of Source Components

Some large documents and files for installation of COTS software were bulky to manage as regular components and were also infrequently accessed and revised. These files were segregated into separate subtrees of the source. In all, the top level has three subdirectories:

- *BT/src/* - the source subtrees for the components
- *BT/doc/* - bulky and multi-component documentation
- *BT/external/* - COTS software support

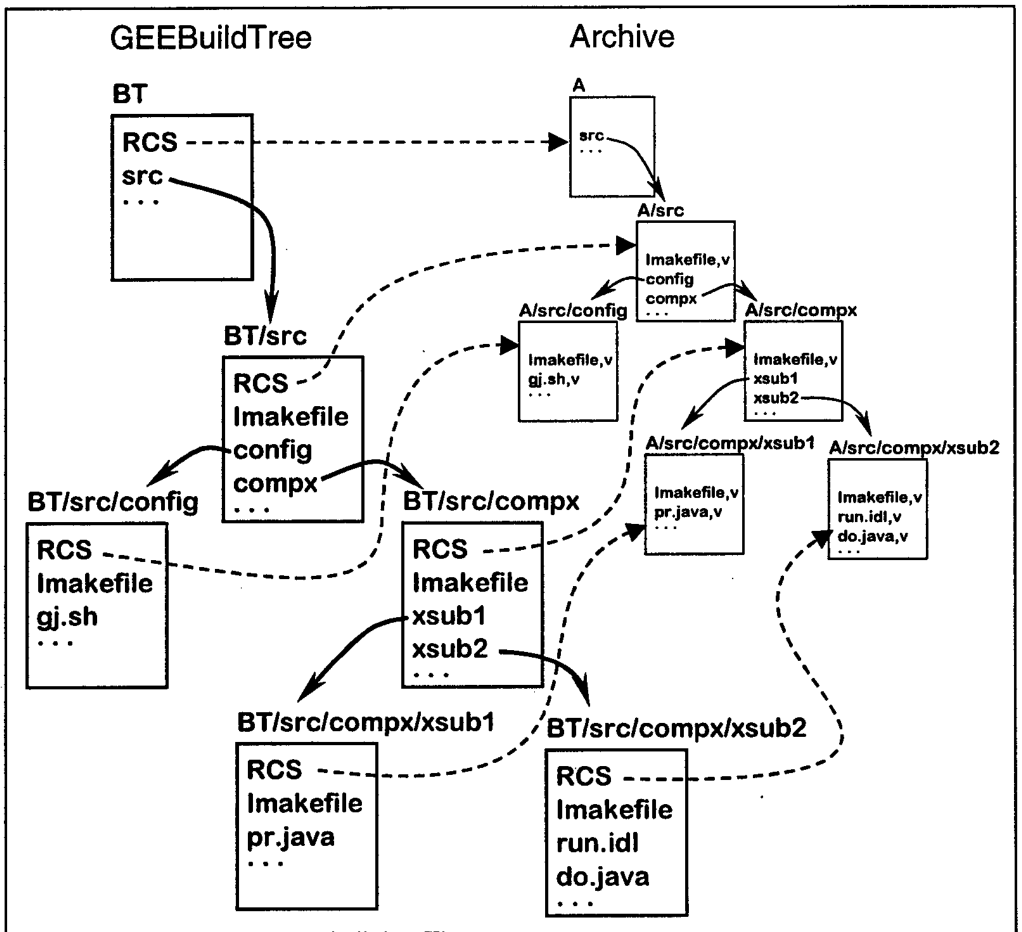
Only the first of these, *BT/src/*, is actively modified and tested by developers. Rebuilding the system can be accomplished by building and installing from this subtree alone. The *BT/doc/* tree is installed every time a full release is made to system directories. The *BT/external/* tree has one subdirectory for each COTS product: Java, Oracle, Visibroker, and JRun. These directories contain instructions for installing new versions of the product and are used primarily when a vendor releases a new version.

From the various tools available for source archive and version control, we chose the *rcs* tools because we were familiar with them and they are mostly satisfactory. A few problems were solved by adding to the tools.

Problem: *Rcs* does not support a tree structure of directories, one for each component. Indeed, the usual approach is that a directory's archive is in a subdirectory named *RCS*. This is unacceptable when the archive is to be shared among a number of programmers.

Solution: The archive is organized as a tree of directories paralleling the source directory structure. A "symbolic link" named *RCS* in each source directory is made to refer to the corresponding directory in the archives.[4]

The resulting directory structure is illustrated in Figure 3. Each *BT/...* directory now has a symbolic link named *RCS*, which points to the corresponding directory in the archive, *A/*.



Dotted arrows are symbolic links. The archived versions of files are in a special format and have names ending in ,v.

Figure 3: Build and Archive Directories: Parallel Trees

[4] This structure with symbolic links called *RCS* was first developed for the Andrew project by David Rosenthal in 1984.

To extract a build directory tree parallel to a source archive tree, we created a script, *rcscotree*. The command

```
rcscotree src
```

creates a subdirectory of the current directory, calls it *./src/*, checks out into it all the files from *A/src/*, and repeats the process recursively to copy files from subdirectories of *A/src/* to subdirectories of *./src/*.[5] To link a single build directory to a source directory, a symbolic link named *RCS* can be created with the UNIX *ln* command or another GEE command, *rcslink*.

In each subdirectory of the checked-out tree, *rcscotree* creates a symbolic link called *RCS* that refers to the archive directory from which the sources were checked out for that directory. As a result, every directory in the source tree appears to have a subdirectory named *RCS*, which is exactly what the normal *rcs* tools expect. When a developer wishes to lock a file for modification, it is checked out and locked with the normal *rcs* command:

```
co -l filename
```

After modification, the file is checked in with a further *rcs* command:

```
ci -u filename
```

Problem: The *ci* command deletes the source tree copy of a file, unless the -u flag is given, and developers frequently omit the -u flag.

Solution: Many techniques have evolved for dealing with this idiosyncrasy of *ci*. Editors have been modified to run *co* automatically, and construction scripts have had rules added to do automatic checkout. The fault is best fixed by providing an alternative to *ci*, one that also solves the next problem.

Problem: Developers fail to note and check all changes to files that they check in.

Solution: We developed *save* as the alternative to *ci*. The command

```
save filename
```

first displays the differences between the previously archived version and the new version. The developer checks these changes while creating the log entry for the new version. Next *save* does a *ci* command and automatically inserts the -u flag so the file remains in the source tree to support future builds. The programmer enters the change log entries as prompted by *ci*.

Problem: Some files in the archive are binary (like Microsoft Word files) and do not have the ASCII line structure assumed by the *rcs* tools.

Solution: Binary files are stored in the archive directly, without using any *rcs* tools; their names lack the trailing "*,v*" which is the hallmark of RCS archive files. Upon checkout with *rcscotree,* these files are directly copied to the source tree. This mechanism is less than adequate; when such a file has been modified, it must be copied into the archive directory and no

[5] The *current* directory when a command is executed has the name "." (dot), and its superordinate directory can be referred to as ".." (dot dot).

version control is done. This lack is partially offset by Microsoft Word's rudimentary form of version control.

2.2 *Make* and *Imake*

Before use, each GEE component must be built, tested, and installed. These operations could be distributed to separate shell scripts, but these would add to the already large number of files to manage and would lack the automatic determination—provided by *make* [Feldman 86]—of the minimal set of actions needed to perform a task. For these reasons and because of its wide availability and familiarity, *make* was chosen for GEE.

In *make*, all operations are described in a file called '*Makefile*.' Each segment of this file denotes a dependency and its actions:

```
target-file: source-file(s)
    actions to build target from source-file(s)
```

When *make* is invoked, the target file is checked to see if it is non-existent or older than at least one of the listed source file(s). In either case, the actions (which are on subsequent indented lines) are executed and are assumed to produce a newer version of target file. The actions are passed to a shell, so there is no limit on what can be done. In GEE many actions are simple shell commands, while others are C programs or scripts for *sh*, *csh*, *awk*, or *sed*. We did not use Java for scripts because of the high overhead for starting a Java program.

Allowing any program to execute for a *make* action is undeniably powerful, but is not without its limitations. The majority of actions can be carried out by calling standard UNIX programs, but in most major systems constructed atop UNIX some actions are best described by writing little "helper" programs. The difficulty is that each such helper must be individually created, source managed, built, and sometimes installed. Thus, the construction/deployment task is made more complex. There is also a possibility of conflict between the names chosen for the helpers of one system and those of another. The need for helpers could be eliminated if suitable commands could be conveniently written directly in the *Makefile*.

One obvious candidate for an action that *make* ought to provide is that of installing a file. In addition to copying the file, it may also be necessary to delete an old version and set the permissions and ownership of the new file. There is a small UNIX program that does an install; however, in typical UNIX fashion, there are two such programs, both called *install*, but differing in command line syntax. To deal with this surfeit, the location of the *install* command is in *imake.tmpl* and can be overridden from *site.mcr*.

Problem, part 1: It may be necessary to specify different variants of the actions depending on the target hardware or software platform. This was especially the case when developing with C and C++; the languages were standard, but the details of library calls varied widely across UNIX variants.

Problem, part 2: There may be a great deal of repetition when the same or similar actions are required for multiple target files.

Solution: Several solutions have evolved for these problems. We cover three in subsequent paragraphs.

The Free Software Foundation's *configure* program [MacKenzie 98] generates a *Makefile* tailored to the capabilities of the hardware and operating system platform for the build and install. It does this by executing test code and building a definitions file based on the result; programs include this file so that they can be written in a platform-independent dialect. *Configure* does not solve the problem of repetition in *Makefiles*, nor was it appropriate for GEE because it is mainly aimed at C, whereas GEE uses Java.

The *qed/qef* system from IPT Corp. [Tilbrook 96] generates *Makefile*-like control files by macro processing. It also provides a macroprocessor/text-processor as part of the capabilities for actions to be performed. This would eliminate most of the scripts and small programs that GEE executes as actions. It is a disadvantage that *qed/qef* is a proprietary product, but the real reason we did not explore it for GEE was our familiarity with the next option and the belief (incorrect) that that option could be implemented with trivial effort.

The option we did choose was the *imake* processor that accompanies the X Windows System [Fulton 89, Dubois 96]. This system was designed precisely to solve the two problems of platform variability and repetition. At heart, the *imake* command itself does nothing more than invoke the C preprocessor[6] on a specified template file. At appropriate places, this template has instructions to copy into its text

- GEE-specific definition files
- a macro definition file
- a directory-specific file called *Imakefile*

For each source file or collection of similar source files, the *Imakefile* invokes one of the defined macros, which then expands to an entire *make* rule including target file, source file(s), and actions. The resulting file is a *Makefile*. It may have repetitions and platform dependencies, but all arise from macro expansions. In general, an *Imakefile* is a small fraction of the size of the corresponding *Makefile*. *Imake* solves the platform dependencies problem by having alternate versions of the definition files for each platform.

When there are subordinate component directories, the *Imakefile* in the parent directory contains instructions that perform recursive *makes* of all subordinate components. A subset build is accomplished by using *rcscotree* to make a mirror of some subset of the Source Archive tree instead of the whole tree. Thus, a developer could have a directory anywhere with a symbolic link to (say) *A/src/compA/*.

[6] The C preprocessor scans a file for lines beginning with #include, #define, or #if and for each, respectively, incorporates and scans another source file, defines a variable, or chooses whether to incorporate and scan subsequent lines.

3 GEE Definitions and Makefile Generation

In GEE, construction/deployment scripts are *Makefiles* that are processed by *make*. However, Gee *Makefiles* are not among the sources in the archive, but are instead generated from a stored *Imakefile* by running *imake*, as described below. Most *Imakefiles* are quite short like this one for a directory containing only a Java program in the default package:

```
JavaPackage(.)
InstallJavaPackage(., ${GEECLASSESDIR})
```

The first line specifies compilation of all Java source files in ".", the current directory. The second line specifies that the resulting *.class* files are to be installed in the directory where all GEE class files are installed.

For another example, the directory *GEEBuildTree/src/GenoServlet/public_html/* installs Web pages. Its *Imakefile* reads

```
ExpandGeeProperties(GeeLogin.h, GeeLogin.html)
ExpandGeeProperties(CstQTrack.h, CstQTrack.html)
MkdirTarget(${GEEHTMLDIR}/GenoServlet)
InstallDataFiles(ALLFILES.html, ${GEEHTMLDIR}/GenoServlet)
```

The first two lines process *.h* files to replace property names with values and produce *.html* files; the third line ensures the existence of a directory in the destination tree; and the fourth line installs the processed files into that directory.

In addition to the *Imakefile*, creation of a *Makefile* uses these definition files:

- *gee.properties*, global definitions for Java: defines gee.destination.directory, names for COTS products directories, and other properties
- *gee.rls*, macro definitions: defines JavaPackage and other macros
- *imake.tmpl*, the template *Makefile*: defines GEECLASSESDIR, GEEHTMLDIR, and other values
- *site.mcr*, a site-specific set of *make* definitions

These files are all in the *GEEBuildTree/src/config/* directory and are described in the next three sections.

3.1 gee.properties

To support building GEE on multiple platforms, at foreign sites, or with different parameters, all environmental dependencies are segregated in a small number of files that are referenced from everywhere else in the system. The first of these, *gee.properties*, is a Java "properties" file; each entry consists of a property name, an equal sign, and a string giving the value for the property. See Table 1 for an excerpt from the *gee.properties* file.

In the table, the first three properties are for GEE as a whole. When the fourth—for the location of the certificate file—is fetched, the string ${gee.destination.directory} is replaced with the proper value from the first property. All instances of ${...} are similarly replaced with the value named by the contents. Each COTS product has a section like those shown for Oracle and Java. There is at least a property naming the directory where the COTS product is installed. Among other properties, Java has the one shown for the classpath. Via various tricks, this one classpath list is used at every point where Java files are compiled or executed.

```
# Overall GEE properties
gee.destination.directory =\                        #root of GEE installation
    /usr/local/gee/dest
gee.version = 03                                    # current GEE version
gee.web.host = gc.sei.cmu.edu                       # host for gee web site

# Certificate file access
gee.keystore = ${gee.destination.directory}\        # keystore file location
                    /etc/oscStore-v${gee.version}
gee.password.for.keystore = 08j06a96s               # keystore password
# Oracle database
gee.oracle.base = /u01/app/oracle                   # root of all Oracle files
gee.oracle.home = ${gee.oracle.base}/\              # Oracle version in use
            product/8.0.3
# JAVA
gee.java.home = /usr/local/jdk1.2                   #where Java is installed
gee.java.classpath =\
    ${gee.destination.directory}/classes:\          # directories with .class,
    ${gee.oracle.base}/product/8.0.3/\              # .jar, and .zip files
            jdbc/lib/classes111.zip:\
    ${gee.java.home}/lib/jsdk.jar:\
    ${gee.java.home}/lib/classes.zip

Comments are in italics; in actuality, comments must occupy lines of their own.
```

Table 1: Excerpt from gee.properties

Some values from *gee.properties* are also needed in programs or files written in other languages, such as HTML, C, or the shell language for *csh* or *sh*. To provide access, a small tool called *Defines.java* was written to read *gee.properties* and produce two other files, *gee.h* and *gee.sed*. The resulting *gee.h* is a C includes file having a definition like

```
#define property-name value
```

for each property. Similarly, *gee.sed* is a control file for *sed* and will replace each property name with its expanded value. (Original versions of *sed* cannot limit replacements to whole words; for this reason, no property name can be a substring of any other property name.)

The main loop of *Defines.java*, shown in Table 2, iterates through the properties and for each property writes one line to each of the two output files. The property names are converted to upper case and have their periods replaced with underlines. Values are processed with the expand function to recursively replace all instances of ${xyz...} with the value of the property named xyz.... (This same function is called by the GEE properties access class, GeeProperties, to expand property values before they are used in Java programs.) For *sed.h*, the chooseDelim function chooses a graphic character that does not appear in the property value. Sample input and output files are shown in Table 3.

```
// Process gee.properties; create gee.h & gee.sed
java.util.Properties props = new java.util.Properties();
java.io.BufferedWriter hFile = <gee.h>;                     // gee.h output file
java.io.BufferedWriter sedFile = <gee.sed>;                 // gee.sed output file
props.load(<gee.properties>);                               // the input file
java.util.Enumeration pe = props.keys();                    // iterate thru properties
while (pe.hasMoreElements()) {
    String key = (String)pe.nextElement();                  // get property name
    String keyOut = key.replace('.', '_')                   // period -> underline
            .toUpperCase();
    String value = expand((String)props.get(key), 0);       // expand property value
    char delim = chooseDelim(keyOut+value);                 // choose delim for sed
    hFile.write("#define " + keyOut + " " + value);         // write line to gee.h
    hFile.newline();
    sedFile.write("s" + delim + keyOut                      // write line to gee.sed
                    + delim + value + delim + "g");
    sedFile.newline();
}
```

Table 2: Defines.java

It is a violation of the principle of "Defer binding" to use values from *gee.properties* via the C preprocessor or *sed*. This violation means that a system rebuild is necessary when these environment values change, as may happen when trying to move the installed files to a different directory. One alternative would be a runtime table-lookup every time a value is used from *gee.properties*. This would make the system bulkier, slower, and more complex. Since many of the affected files are ASCII text—HTML or shell scripts—a second alternative is to write a program to revise these files when the system is relocated. The most accurate approach would be to generate a list of files during system build and then process them for relocation. We have not explored these alternatives.

gee.properties – *input*

```
gee.destination.directory =/usr/gee/dest
gee.version = 03
```

gee.h – *output*

```
#define GEE_DESTINATION_DIRECTORY /usr/gee/dest
#define GEE_VERSION 03
```

gee.sed – *output*

```
s:GEE_DESTINATION_DIRECTORY:/usr/gee/dest:g
s/GEE_VERSION/03/g
```

The input lines shown for gee.properties *are transformed by* Defines.java *into those shown for* gee.h *and* gee.sed. *Note that two different delimiters, colon and slash, are used in the output for* gee.sed.

Table 3: Inputs and Outputs of Defines.java

3.2 The *imake* Template, *imake.tmpl*

Imake invokes the C preprocessor on a template file; for GEE an abstract of that file, called *imake.tmpl*, is shown as Table 4. Variables defined with an equal sign are *make* variables and are later referenced with the syntax ${*makevar*} as in ${TOUCH}. Other capital letter variables like GEE_DESTINATION_DIRECTORY are #defined in *gee.h* as a result of being defined in *gee.properties*. The template defines the *make* variables GEEBINDIR, GEECLASSESDIR, and others for use as the destination in InstallXxxx macro calls; these variables also provide an additional level of indirection so each class of files can be independently stored in a separate location.

The #include lines incorporate the various GEE definition files. The lines with double colons ensure that the principal build and install targets are always defined, even if nothing needs to be done in this directory. The convention is that the capitalized *make* targets such as Compiles and Installs are recursive and perform their function in this and subordinate directories; lower case targets operate only in the current directory. Note that making the target Install will do install in each directory and the latter will first make compile; consequently, all compilations are up-to-date for each Install.

Problem: After developing the build system, it failed its first tests by others. The problem was that some developers have non-standard command environments: their own aliases for commands, strange paths with command variants, and other anomalies. It is also a problem that some UNIX systems have different commands for certain functions.

Solution: To provide for adaptation to UNIX system differences, the template defines a *make* variable for each UNIX command, and these variables are used in "actions to build target from source-file(s)." For example, the value from SED=/bin/sed is used in the macro to convert files with gee.sed:

```
${SED} –f gee.sed infile > outfile
```

To avoid command aliases defined in developer environments, the commands are specified in the definitions with their full pathnames. The inclusion of "*site.mcr*" lets a site define variables for *make* without defining them in *gee.properties*. (Note that definitions in *site.mcr* must be in *Makefile* syntax and cannot be used in any other type of file. Many *gee.properties* lines define directory locations; these could be moved to *site.mcr*. At present, they remain where they originally were placed.)

```
#include <gee.h>
/* variables for GEE destination directories */
    GEEBINDIR = GEE_DESTINATION_DIRECTORY/bin
    GEECLASSESDIR = GEE_DESTINATION_DIRECTORY/classes
    ...
/* Variables as names for the UNIX commands */
    AWK = /bin/awk
    CAT = /bin/cat
    TOUCH = /bin/touch
    ...
#include "gee.rls"
#include "site.mcr"
/* Define universal targets in case Imakefile does not */
all:: compile
Compiles:: compile
compile::
Installs:: install
install:: compile install.time
...
/* incorporate the directory specific Imakefile */
#include INCLUDE_IMAKEFILE
install.time::
    ${TOUCH} install.time
```

Note that this template file #includes gee.h, gee.rls, site.mcr, *and* INCLUDE_IMAKEFILE. *The last of these is defined to the macro preprocessor so it refers to the* Imakefile *in the current directory.*

Table 4: The imake *Template File,* imake.tmpl

3.3 gee.rls

The *gee.rls* file defines the macros invoked by *Imakefiles*. For instance, the *Imakefile* line

```
    MkdirTarget(${GEELIBDIR}/images)
```

invokes the following definition:

```
    #define MkdirTarget(dirs)
    install.time:: makedirs
    makedirs::
        for i in dirs; do \
            ${CONFIGDIR}/geemkdirs $$i;\
        done;
```

When the invocation is expanded according to this definition, the resulting *Makefile* will be something like

```
    install.time:: makedirs
    makedirs::
        for i in ${GEELIBDIR}/images; do \
            ${CONFIGDIR}/geemkdirs $$i;\
        done;
```

Both GEELIBDIR and CONFIGDIR are given values in *imake.tmpl.* The *geemkdirs* command is installed from the *GEEBuildTree/config/* directory; it creates a directory and any parent directories that do not yet exist.

In the above example and throughout the body of this report, macro parameter names are italicized and the following aspects have been omitted from macro definitions:

- the first line of each definition, which shows in the *Makefile* an image of the invoking macro from the *Imakefile*
- the special end-of-line delimiters @@\, which hide line breaks in the macro definition, but become line breaks in the *Makefile*
- lines that merely print messages in the log file of the build
- rules and rule fragments dealing with deletion of files for the make target 'clean'
- features dealing with error handling: @(...), NOSHERRORS, SHVERBOSE, || exit 1
- commands for building GEE, which does a build and install of everything
- the make variable MFLAGS, which passes command line flags to subordinate *makes*

None of these omissions have been made in Appendix C, which can be consulted for details.

Reading *gee.rls,* you will find the curious definition

```
    #define ALLFILES *
```

This and other similar definitions avoid various restrictions. ALLFILES, for instance, is used in rules where an asterisk would be written after a slash, as in "dirx/*.java."[7] Otherwise, a bald "/*" in the string would signal the start of a comment to the C preprocessor, and everything up to the next */ would be omitted from the generated *Makefile*.

The rule to use *gee.sed* to expand property names is ExpandGeeProperties:

```
#define ExpandGeeProperties(infile, outfile)
compile:: outfile
outfile: infile
    ${SED} -f ${CONFIGDIR}/gee.sed infile > outfile
```

Thus ExpandGeeProperties(genmake.csh, genmake) will produce a rule that—whenever *genmake* is non-existent or older than *genmake.csh*—will run *sed* to process *genmake.csh* using the *gee.sed* script and producing *genmake*. In particular, this will replace the value of GEE_DESTINATION_DIRECTORY in *genmake.csh* with its actual value from *gee.properties*.

Other portions of *gee.rls* will be described in subsequent sections.

3.3.1 gee.rls: Macro Rules for Operations on Entire Subtrees

To build an entire subtree, *Makefiles* in parent directories have rules for performing *makes* on their subordinate directories. There is a convention for this: an *Imakefile* defines the *make* variable SUBDIRS to be a list of the subdirectories and then includes an invocation of the RecursiveMakes rule. At the top level, this looks like the following:

```
SUBDIRS = config gee rdbms security naming busLog appclient
RecursiveMakes(${SUBDIRS})
```

Because SUBDIRS is defined, its definition can be overridden when initiating a *make*. The following shell command makes the Compiles target in the current directory and each of the three named subdirectories:

```
make Compiles SUBDIRS='config gee rdbms'
```

The definition of RecursiveMakes directly calls five other macros:

```
#define RecursiveMakes(subdirs)
CompileSubdirs(subdirs)
InstallSubdirs(subdirs)
JavalistSubdirs(subdirs)
GEESubdirs(subdirs)
SystemInstallSubdirs(subdirs)
```

[7] An asterisk in a file name for the shell matches any sequence of characters other than slash. The string "/*" is treated specially by *imake* because it is the C comment delimiter, and *imake* uses the C preprocessor.

Thus RecursiveMakes passes its subdirs list to each of five macros, four of which are defined something like this:

```
#define CompileSubdirs(dirs)
SubdirTarget(dirs,Compiles,"compiling")
```

In other words, it invokes the SubdirTarget macro, passing along the directory and introducing the target to be built (and a comment string for the build log). The latter macro is defined like this:

```
#define SubdirTarget(dirs, mktag, string)
mktag::
    for i in dirs; do \
            (cd $$i; ${MAKE} CONFIGDIR=${CONFIGDIR} mktag); \
    done
```

For CompileSubdirs, the expansion of SubdirTarget will define the target Compile. The action for this target is to iterate through the list dirs of subdirectories and make the Compile target with each as the current directory. Note that the value of CONFIGDIR is passed along to the subordinate *make*; this is explained in Section 3.4.

Since *make* needs a *Makefile*, the above approach cannot generate *Makefiles* throughout a tree. Instead, the system uses a more intricate mechanism involving the GEESubdirs() rule; see Section 3.4 for the details.

3.3.2 gee.rls: Installation Rules

Installation of files from each component is specified in its *Imakefile*. For instance, these lines appear in *GEEBuildTree/src/busLog/Imakefile*:

```
InstallJavaPackage(., ${GEECLASSESDIR})
InstallIDLSource(Genotype.idl, ${GEELIBDIR}/idl)
InstallIDLModule(Genotype, ${GEECLASSESDIR})
InstallJavaPackage(Genotype/UserFactoryPackage, ${GEECLASSESDIR})
```

Most of the InstallXxxx macros are defined like this one for InstallDataFiles:

```
#define InstallDataFiles(files, dest)
InstallFiles(files, ${INSTDATAFLAGS}, dest)
```

with each having different INSTxxxxFLAGS. These flags are defined in *imake.tmpl* and set the modes of the installed file—read-only or not, executable or not. The internal macro InstallFiles is defined this way:

```
#define InstallFiles(files, mode, dest)
install.time:: files
    for i in `${ECHO} '$?'`; do \
        ${INSTALL} mode $$i dest/$$i; done
```

This specifies that when the target install.time is older than one or more of the files, each younger file will be the subject of a ${INSTALL} command using INSTxxxxFLAGS as the mode and installing the file under its own name into the given destination directory. The curious `` `echo '$?'` `` deals with cases where a filename contains a dollar sign.[8] With $? alone, a dollar sign in the value is interpreted by the shell to be a reference to a (probably undefined) shell variable. The inner apostrophe quotes protect the dollar signs from treatment as a shell variable but also make the value a single shell "word." The invocation of *echo* causes that word to be expanded back into a space-delimited list of file names.

The usual mode of operation with *make* is to have a target file depend on some source files and, if a source is newer, to execute an action; presumably, the action generates that target file from the sources. The approach can be made to work for file installation: the target file would be the destination file, the source would be the file to be installed, and the action would copy source to target. GEE installation differs; in GEE the target for installation is a "dummy" file called *install.time*. Every installation macro generates a rule dependent on *install.time*. This rule performs the installation if *install.time* is non-existent or is older than one of the files to be installed. Near the end of *imake.tmpl* is a rule for *install.time* that touches the dummy file if all prior *install* actions have succeeded. This scheme has the advantage of not requiring one rule per file and of permitting a reinstallation by deleting all instances of *install.time*.

There are, however, a number of disadvantages to dummy files. They are often given names with leading periods. Since such files do not normally appear in directory listings, it is easy to miss seeing one that should be manually deleted, inserted, or modified. The action can be triggered only by deleting the dummy file; in the case of *install.time*, if the target file is deleted it will not be reinstalled as long as *install.time* is unchanged. When installations are occurring from multiple build trees by multiple developers, the dummy files may exist in one tree and not in another, again causing incorrect results. In a future effort, GEE's install mechanism should be redesigned.

GEE already provides a rule where the installation depends on the date stamps of the two files. It is defined thus:

```
#define InstallFileAsNeeded(file, mode, dest)
install::dest
dest: file
        ${INSTALL} mode file dest
```

If *dest* is older than *file*, the installation is done. (Note that because *dest* must appear at the beginning of a line, it is not acceptable to have spaces after the second comma in an invocation of InstallFileAsNeeded.)

[8] In this report, grave accents (‘, which appear on American computers as little-6 single-quotation marks and are sometimes called backwards apostrophes) are emphasized to distinguish them from apostrophes (', which appear as little-9 single-quotation marks). When interpreting a text command, the shell treats each pair of grave accents as containing a subordinate command. The subordinate is executed, and its output replaces the grave accents and their contents.

An important special case of InstallFileAsNeeded is for installation of files in system directories. In this case, the installation is usually done as the privileged, "root" user because only that user has write access to the necessary directories. However, if care is not taken, the root user will be the owner of the installed file. In most cases, it is more appropriate that the owner be the same as the owner of the destination directory. The SystemInstallFileAsNeeded macro achieves this ownership as shown here:

```
#define SystemInstallFileAsNeeded(file, mode, dest)
systeminstall:: dest/file
dest/file: file
        ${INSTALL} mode file dest
        (df=dest/file; \
        cmd=`${LSLD} dest | ${AWK} '/^d/\
            {print "${CHOWN} " $$3 " '"$$df"';";\
             print "${CHGRP} " $$4 " '"$$df"'"}'`;\
        ${SH} -c "$$cmd")
```

After using the *install* command to install the file, this macro changes its group and ownership to that of the directory into which the file is installed. This second operation occupies all of the last five lines of the macro and is especially intricate because the C preprocessor, *make*, the shells, and *awk* have overlapping quote characters and different meanings for '$'. The first of these lines defines df as a shell variable having the destination filename as its value. It is invoked in the third and fourth lines as '$$df' using the shell variable syntax; the two dollar signs are converted to one by *make* before passing the command to the shell. The line beginning cmd=... runs 'ls -ld' on the destination directory and pipes the output through *awk* to generate a shell command that is executed by the last line. The '$$3' and '$$4' references are in *awk* syntax and select the third and fourth entries from the result of the 'ls –ld'. It may be instructional to check for yourself that each and every quote mark and dollar sign is essential.

In most cases in GEE, installation actions could be specified in the rule that creates the file. For instance, many GEE *Imakefiles* have these two lines:

```
JavaPackage(.)
InstallJavaPackage(., ${GEECLASSESDIR})
```

Since every package is installed in GEECLASSESDIR, the installation rule could be part of the rule for JavaPackage. That it was not so done is primarily a vestige of the generality of the Andrew script mechanisms.

3.4 Generating the *Makefile* with *imake* and *genmake*

To build in a directory, the *Imakefile* must be macro-expanded into a *Makefile*, using the various definition files described above. In the build tree, these files are in *GEEBuildTree/src/config/*, but they are also needed in an installed system to support the subset build option of Figure 1. For this purpose, let us suppose that GEE has been installed into the directory given by shell variable GEEDIR. As part of that installation, the *GEEBuild-*

Tree/src/config/ files are copied into $GEEDIR*/config/*, and a command called *genmake* is installed in $GEEDIR*/bin/*.

A *Makefile* can always be derived from an *Imakefile* with the *imake* command, but this involves getting a lengthy set of parameters correct, especially the location of the definition files, either *GEEBuildTree/src/config/* or $GEEDIR*/config/*. To reduce typing and its potential for errors, various shortcut alternatives to *imake* are provided, depending on how far along the GEE installation has progressed. In all cases, the trick is for the command to determine the location of the correct *config/* files. Inside the *Makefile*, this location will be the value of the *make* variable CONFIGDIR. There are four cases:

Case 1. A *Makefile* already exists, but a new one is needed because the *Imakefile* has changed. To do the reconstruction, the command is

```
make Makefile
```

This command invokes the shell command in ${IMAKE_CMD}, which has this value:

```
imake -Timake.tmpl -I. -I${CONFIGDIR} \
          -DCONFIGDIRDEFN=${CONFIGDIR}
```

This action runs *imake* specifying that the template file is *imake.tmpl*, the include directories are . and ${CONFIGDIR}, and the value of CONFIGDIRDEFN is ${CONFIGDIR}. In this case, the value of CONFIGDIR from the current *Makefile* is passed along and will become the value in the new *Makefile* because *imake.tmpl* has these early lines:

```
CONFIGDIR = CONFIGDIRDEFN
#undef CONFIGDIRDEFN
```

Since CONFIGDIRDEFN has a value known to the preprocessor, its instance is replaced in the generated *Makefile* with the value from the command line, which was the value in the original *Makefile*.

Case 2. A *Makefile* exists and *Makefiles* are needed for all subordinate directories. The command is

```
make Makefiles
```

(note the trailing "s".) This command invokes a rule where the target Makefiles depends on making the target subMakefiles. The rule for subMakefiles does not depend on anything, so its actions are always executed; it iterates through the SUBDIRS list (Section 3.3.1), sets $$i to the name of each subdirectory in turn, and for each executes two *make* commands:

```
make subdirMakefile  CONFIGDIR=${CONFIGDIR} MAKE_SUBDIR=$$i
(cd $$i; make CONFIGDIR=${CONFIGDIR} subMakefiles)
```

The first command makes the target subdirMakefile, which creates a *Makefile* in the subordinate directory by using the same ${IMAKE_CMD} as in Case 1. The second command changes directory to the subordinate and makes subMakefiles again to recursively generate *subsub...Makefiles*. In either case, the actual creation of a *Makefile* is done in the rule for subdirMakefile; since it uses the ${IMAKE_CMD}, the value of CONFIGDIR from the present *Makefile* is passed along to become the value in the new *Makefile* (by virtue of the –DCONFIGDIRDEFN in ${IMAKE_CMD}).

Case 3. No *Makefile* exists, but GEE has been installed. Part of that installation, we noted earlier, was installation of the *genmake* command and installation of copies of the definition files in $GEEDIR*/config/*. In any directory where there is an *Imakefile*, the command

```
genmake
```

will create a *Makefile*. As shown in Appendix G, *genmake* essentially executes the command

```
imake -T imake.tmpl -I${TARGET} -I${TARGET}/config \
      -DCONFIGDIRDEFN=${TARGET}/config
```

where TARGET has been given the value of $GEEDIR (or a value from the command line). Note the CONFIGDIRDEFN is set to the *config/* directory used for the expansion, and that directory will be the value for CONFIGDIR in the created *Makefile*.

Case 4. Neither a *Makefile* nor *genmake* exists. When the *src/...* tree is first copied out from the source archive, there are no *Makefiles* and one must be created in the *GEEBuildTree/src/* directory. This is done using *make*, but passing it *GEEBuildTree/src/Imakefile*. Thereafter, Case 2 can be used to generate *Makefiles* in the rest of the tree. In all, the bootstrap process in directory *src/* is these two commands:

```
make -f Imakefile
make Makefiles
```

The file *GEEBuildTree/src/Imakefile*, as shown in Table 5, is special in that it can be used as both an *Imakefile* and a *Makefile*. It can do so because it contains no macros to be expanded by *imake*. The real *Imakefile* is *GEEBuildTree/src/Amakefile*, as shown in Table 6.

```
all:: config/gee.h config/gee.sed Amakefile config/imake.tmpl \
          config/site.mcr config/gee.rls config/geemkdirs
    imake -I. -I./config -Timake.tmpl -f Amakefile \
          -DCONFIGDIRDEFN=‘pwd‘/config
config/geemkdirs:
    cc -o config/geemkdirs config/geemkdirs.c
config/gee.h config/gee.sed: config/gee.properties config/Defines.java
    javac config/Defines.java
    java -classpath config:$$CLASSPATH Defines config/gee.properties
```

Table 5: The Special Top-Level Imakefile

The first rule of Table 5 runs *imake* and specifies with the -f switch that the *Amakefile* is to be read instead of an *Imakefile*. Most of the files on which dependencies are declared are pre-existing files whose change will trigger a rebuild; however, the first three listed depend on files that are created by the *make* process according to the second and third rules. The second rule creates *geemkdirs*, a small program that creates a directory and any absent superordinate directories. The third rule compiles *Defines.java* and runs it to create *gee.h* and *gee.sed*. Note that a C compiler and a Java environment must be accessible from the developer's path before using the command. Otherwise, all GEE development is independent of the developer's path.

```
/* create destination directories */
DIRS = GEE_DESTINATION_DIRECTORY ${GEEBINDIR} ${GEELIBDIR} \
    GEE_DESTINATION_DIRECTORY/config  ${GEECLASSESDIR} \
    ${GEEETCDIR} ${GEEDOCDIR} ${GEEHTMLDIR} ${GEELIBDIR}/idl
MkdirTarget(${DIRS})
/* make all subdirectories */
SUBDIRS = config gee rdbms security naming busLog appclient
RecursiveMakes(${SUBDIRS})
/* build descriptions for all classes */
Javadocs:: install.time javadoc/index.html
    ${RM} -rf ${GEEHTMLDIR}/javadoc
    ${TAR} cf - ./javadoc | (cd ${GEEHTMLDIR}; tar xf -)
```

Table 6: Amakefile *(The Top-Level* Imakefile*)*

The effective top-level *Imakefile* in Table 6 creates destination directories, provides for recursive makes of all subcomponent directories, and installs Java class documentation in the destination when Javadocs is made.

Problem: Some users are accustomed to X Windows and use its *xmkmf* command to start building the system.

Solution: The top-level *Imakefile* works as an *Imakefile* for *xmkmf* and invokes the same steps as when it is used as a *Makefile*.

4 Java Challenges

Java is a significant advance in programming languages not because it introduced remarkable new features, but because it carefully selected and combined the most useful features from other languages. Similar care has produced the many Java application programmer interfaces from available concepts. It is therefore disappointing that no such simplification has occurred for Java construction and deployment. The Java developers themselves use the *gnumake* version of *make* [Sun 99b]. Various Java development environments are marketed, but they are limited in their capabilities when the system incorporates non-Java source files and COTS products.

Java poses a number of challenges for traditional development tools like *make* and *imake*. These were compounded for GEE development, which took place during the transition of Java from version 1.1 to 1.2.

4.1 Finding gee.properties; Preprocessing Java Files

Since premature binding of values to identifiers limits the flexibility of a system, GEE uses the *gee.properties* file to defer binding until as late as execution time. The GeeProperties class implements access to this file for all Java classes.[9] This mechanism avoids one of the reasons that justifies preprocessing for C files. However, there is still a need to preprocess one *.java* source file:

Problem: How does GEE find *gee.properties*?

Solution: The recommended Java solution for finding an initial properties file is to pass its location as a parameter on the command that invokes the system. This seemed to us a burden for most users. The usual solutions for UNIX and Windows are to bind a name to the location in the environment or the registry, respectively. However, Java programs cannot readily access such values. The GEE solution is to preprocess the source code of *GeeProperties.java* so it contains the location of the GEE installation directory. In practice, the source file *GeeProperties.j* is preprocessed with ExpandGeeProperties to replace GEE_DESTINATION_DIRECTORY with the value from *gee.properties*. This produces the file *GeeProperties.java,* which is then compiled and installed like any other Java source file.

The above "solution" solves nothing if the GEE objects are moved to a directory other than the one specified in the build-time *gee.properties*. After such a move, either *GeeProperties*

[9] Source files that are preprocessed with gee.sed (Section 3.1) are bound at build time. Most of these could be converted to runtime lookups, but this would complicate the source code.

must be rebuilt and re-installed, or the location must be specified on the command line after all.

4.2 Locations for Java Files

A Java "package" is a set of classes that together implement some subsystem; typically a GEE component has sources for one or more Java packages. Where should they be placed? For components written in C and C++, the source and object files are in the component's principal directory. This scheme fails for Java packages due to various conventions. First, the source files of a package each contain the name of the package as a name segmented with dots as in "edu.cmu.sei.gee." Second, by convention, Java source files are stored in a directory whose pathname ends with the package name, except that dots are replaced with slashes: *edu/cmu/sei/gee*; we call this the "slashed package name." Third, the Java compiler places the generated *.class* files for a package into a directory whose pathname also ends with the slashed package name. These naming conventions conflict with the GEE component directory structure. After study, we decided that when GEE sources include a Java package, the component's directory has a subdirectory with the slashed package name as its path, and both the Java sources and objects are stored in this directory.

A number of other options for storing Java source and object files were considered. For instance, it might have been preferable to have the source files directly in the component directory. This would have worked with the Java compiler, but it conflicted with conventions expected by some other Java tools. Another option was to separate the Java sources into a separate package-based directory tree, thus losing the management advantages of components. Various other tricks using symbolic links were rejected to avoid complexity.

Chapter 3 gave examples of invocations of the JavaPackage and InstallJavaPackage rules using a single dot as the first argument. This dot corresponds to compiling the "default" Java package—that is, source files that do not specify themselves as part of any package. When source files are in a package, the slashed package name must appear as the first argument to the macros. For the package edu.cmu.sei.gee, complete calls would appear in the Imakefile as

```
    JavaPackage(edu/cmu/sei/gee)
    InstallJavaPackage(edu/cmu/sei/gee, ${GEECLASSESDIR})
```

The definitions of the rules invoked by these macro calls are provided in the next section.

In passing, note that the Java package and class naming conventions violate the second principle, Defer Binding. Since a portion of the filename appears within the file, the binding is made at the moment of birth. Changing a package name means changing both the file contents and its location. Indeed, yoking name and location has proven troublesome in cases where a package name must change; all files importing that package must be modified to refer to the package by its new name.

4.3 Choosing What to Compile

The first conflict between Java and *make* is in the area of choosing what source files need to be compiled. The standard *make* dependency scheme checks whether a source file is newer than the corresponding compiler output and, if so, recompiles the source file. The compiler is invoked separately for each changed source file. Unfortunately, mutual dependencies between Java classes can mean that *no* sequence of separate compilations can successfully compile a Java package. In view of this, the Java compiler permits multiple simultaneous compilations, which not only solves the dependencies problem but is also considerably faster because it avoids the high overhead of starting the compiler.

Problem: How do we generate a list of Java source files that need to be recompiled?

Solution: One solution would be to recompile every source file for every build, but this would be onerous for small changes. For GEE we implemented a small C program, *needjc* (Appendix H), which checks a list of Java class files and determines which corresponding source files need to be recompiled. A third solution would use a dummy file like ".compiled" that retains the date of the last compilation. By having this dummy as the target and the list of .java files as sources, a make rule could remake only those files changed since the time of .compiled. This approach suffers all the disadvantages of dummy files noted in Section 3.2.2.

Problem: When a package changes, any other package that imports it may have to be changed as well. How can *make* determine what needs to be recompiled due to changes in packages?

Solution: There is no simple solution. A Java source file may declare what packages it uses, but fully qualified names may be used instead; moreover, no qualification is needed to refer to another class in the same package. These provisions mean that the source must be fully parsed—including name resolution—to locate interpackage calls; this is a processing effort far beyond the simple timestamp processing of *make*. Fortunately, when package sources are changed in a way that requires recompilation, the signature of method calls will have changed, and source files that call the package should also be changed. If they are changed, they will be recompiled by a *make*; if not, a runtime Java error occurs. When this has happened and after the sources are appropriately revised, it is usually safest to do a complete rebuild of the system. (A new installation from source will have a complete recompilation, so the runtime error cannot occur in this case.)

4.4 Compilation and Execution

Java executions need access to the files containing the compiled *.class* files for all packages used. Since these files may be scattered throughout the file system, the Java executor expects to find the directories listed either on its command line or in an environment variable called CLASSPATH. This same list of files and directories is also necessary for compilation so the Java compiler can check calls to external packages.

For simple Java programs, there is little difficulty in creating a CLASSPATH value. GEE, however, uses a considerable variety of external packages, including one for each COTS product, and some effort was needed to determine what and where they were. This effort was recorded in gee.java.classpath of *gee.properties* from which it is available throughout GEE. Its value is this:

```
    gee.java.classpath =\
        ${gee.destination.directory}/classes:\
        ${gee.oracle.home}/jdbc/lib/classes111.zip:\
        ${gee.java.home}/lib/jsdk.jar:\
        ${gee.java.home}/lib/classes.zip
```

Problem: How can the CLASSPATH from *gee.properties* be used for Java executions initiated by shell commands instead of with *make*? The problem is exacerbated by the fact that different developers each have their own scripts for establishing access to Java, and these may differ in their CLASSPATHs and may not include some GEE libraries.

Solution: For GEE, special Java invocation commands were installed: *gjc* for Java compilation and *gj* for initiating a Java program. As these commands are installed, they are preprocessed from *gee.properties* and have the full and correct GEE CLASSPATH. Developers can augment this CLASSPATH by setting a CLASSPATH in the environment before running *gj* or *gjc*. The script for gjc is this:

```
    #!/bin/sh
    cp=${CLASSPATH:-}${CLASSPATH:+:}GEE_JAVA_CLASSPATH
    GEE_JAVA_HOME/bin/javac \
        -bootclasspath GEE_JAVA_BOOTCLASSPATH  \
        -classpath ${cp} ${GJCOPTIONS} $*
```

The two instances of CLASSPATH check for its existence and, if it exists, they insert the value and a colon. The *javac* compiler is invoked with its full pathname and is passed both a classpath and a bootclasspath (for which, see the discussion of VisiBroker in Section 5.3).

Problem: Developers usually have a CLASSPATH that is more extensive than is needed to augment the GEE CLASSPATH.

Solution: None. The result is that the CLASSPATH is much bigger than needed, but this does not bother Java. It seemed best not to use a separate shell variable for augmenting the CLASSPATH to *gj* and *gjc* because that would require additional developer training.

Problem: When constructing GEE from sources in a sandbox, it is usually desirable to use *.class* files from sibling directories in the sandbox rather than those from the regular CLASSPATH of installed *.class* files.

Solution: An additional *make* variable, EXTRACLASSES, is incorporated into the CLASSPATH when Java is invoked from an *Imakefile*. This appears as the first line of the second *Imakefile* example in Chapter 3:

```
EXTRACLASSES = ../gee:../rdbms:../security:../naming
```

In this case, the *.class* files from four sibling directories are used and will supersede the same classes that may exist in installed libraries.

In addition to EXTRACLASSES, a site may want to incorporate libraries without modifying *gee.properties*. These can be defined in *site.mcr* as the value of the *make* variable SITE_CLASSES. Given these and the above, here is the definition of the Java compilation command as it appears in *imake.tmpl*:

```
CLASSES = .:${EXTRACLASSES}:${SITE_CLASSES}
JAVACOMPILE = CLASSPATH=${CLASSES} ${CONFIGDIR}/gjc
```

This command is invoked in the fifth line of this *gee.rls* macro for building a Java package:

```
#define JavaPackage(pkgname)
compile::
    jfiles=`${CONFIGDIR}/needjc pkgname/ALLFILES.java`; \
    if [ "X$${jfiles}X" != "XX" ] ; then \
        ${JAVACOMPILE} $${jfiles}; \
    fi
```

Here we see that *needjc* is called to find out which *.java* files are newer than their *.class* file, and the resulting list is stored as the shell value named jfiles. If jfiles is not the empty string, then ${JAVACOMPILE} is executed for that list of files.

GEE packages are installed by invocation of this macro:

```
#define InstallJavaPackage(package,dir)
MkdirTarget(dir/package)
InstallFiles(package/ALLFILES.class, ${INSTCLASSFLAGS}, dir)
```

When called, the *dir* value is the slashed package name, so the installation directory is a possibly non-existent subdirectory of the installation directory passed as the second argument. The first line of the macro body creates the destination directory, and the second line installs the files. For GEE, the installation directory is always ${GEECLASSESDIR}.

4.5 external/java

The source subtree *GeeBuildTree/external/* contains subdirectories for each COTS product. Since a Java compilation and execution environment must exist before the installation of GEE (for Defines.java), Java is not installed from the *external/* subtree. Nonetheless, there is a *GeeBuildTree/external/java/* directory containing a class library fetched from the Net for jsdk, the Java Servlet Development Kit. This library is installed into the Java library directory where it is expected by *gee.properties*.

4.6 Java Version Changes

Initial GEE development occurred during the latter stages of development of Java 2, the stages where there was roughly one new distribution per month. Two of these were downloaded and used. Because the location of Java and the CLASSPATH value were both in *gee.properties,* it was possible to do most of the transition to a new version of Java with minimal effort. The commands in *GEEBuildTree/external/java/* were used to update the new Java installations to incorporate jsdk.

Unfortunately, new versions of Java changed the locations and names of some vendor-provided packages. It was easy to incorporate the new libraries into the CLASSPATH with the aid of *gee.properties*, but existing GEE source files referred to those packages by their old names. There was no easy repair for this: all such source files had to be changed by hand.

When GEE changed from version 0.1 to 0.2, we wanted to leave the sources of GEE v0.1 unchanged. We could not because it had been built using "*.../jdk1.2/*" as the location of the Java installation, and the new Java had that same name. It was necessary to retain both installations of Java, renaming the older from *jdk1.2* to its full name of *jdk1.2beta4.* Adapting the v0.1 sources was then a simple matter of changing its *gee.properties* to refer to *jdk1.2beta4.* Meanwhile, *gee.properties* for GEE v0.2 was able to use the latest version of Java under the name *jdk1.2.*

4.7 Javadocs

Java's *javadoc* command produces an outstanding set of Web pages describing the interfaces to packages. One tricky part of using *javadoc* for GEE was to figure out how to provide rules in the construction scripts so the Javadoc Web pages could be built for any subtree of the sources, whether for the developer of a single component or as documentation of the entire tree. In our solution, the command

```
make javadoc
```

creates subdirectory *javadoc/* in the current component directory and builds there the Web pages for the Java packages in that single directory. The command

```
make Javadocs
```

with a capital "*J*" and a plural creates the same subdirectory, but puts into it the Web pages for the entire subtree rooted at the current directory. In the top-level *src/* directory, making Javadocs also installs the result into the destination directory tree, again in a subdirectory called *javadoc/.*

The input to *javadoc* is a list of pathnames of *.java* files to be included. To be able to build both local and full subtree versions of the Web pages, it was necessary to generate the list of files in a separate step before running *javadoc*. For Javadocs, this step must recur through the directory subtree. Initial attempts to generate the list with *make* rules having no source files failed because the actions associated with such rules are performed for every make execution. Therefore the *Makefile* entries to generate the list of *.java* files had to have actual files as their sources, and these files had to be created by other rules. The process we finally arrived at begins with a rule in *imake.tmpl* that causes both javadoc and Javdocs to depend upon existence of two files:

```
Javadocs javadoc:: javadoc/,javafiles javadoc/index.html
```

The first of the two source files is the generated list of *.java* files, while the second is a file that is guaranteed to exist after running *javadoc*. Subsequent rules for creating the list have *javadoc/,javafiles* as their target. There are three such rules in *imake.tmpl* (and therefore in every *Makefile*):

```
javadoc/,javafiles::
        ${RM} -rf javadoc
        -${MKDIR} javadoc
        ${TOUCH} javadoc/,javafiles
javadoc/,javafiles:: javalist
javalist::
```

The first of these rules removes any pre-existing *./javadoc/* subdirectory and creates a new one having as its contents an empty file named *,javafiles*. The second says that to create the *,javafiles* file, it is necessary to make the target *javalist*. The third is a dummy rule that guarantees that there will be a rule for *javalist* in every *Makefile*; it does nothing. Actual insertion of names into *,javafiles* is done by an additional dependency in the rule for JavaPackage:

```
javalist::
        (cd ${JAVADOCTOP}; \
        ${LS} -1 ${JAVADOCLIMB}/pkgname/ALLFILES.java) \
                >> ${JAVADOCTOP}/javadoc/,javafiles
```

This command changes the directory to JAVADOCTOP, the directory containing the *javadoc/,javafiles,* to be augmented and runs the *ls* command to list the contents of the directory JAVADOCLIMB, which is the subdirectory of JAVADOCTOP containing packages to be included.

Problem: Get values for the *make* variables JAVADOCTOP and JAVADOCLIMB.

Solution: When creating *javadoc/,javafiles*, in a single component directory, both JAVADOCxxx variables can be "." and these values are established with the following two lines early in *imake.tmpl*:

```
JAVADOCLIMB = .
JAVADOCTOP = .
```

When *,javafiles* are being collected for an entire subtree, the line

```
JavalistSubdirs(dirs)
```

of the RecursiveMakes macro comes into play. It is expanded by the rule for SubdirTarget, which was abbreviated in Section 3.3.2 where the lines for passing JAVADOCTOP and JAVADOCLIMB were omitted. The recursive call on *make* actually looks more like this:

```
${MAKE} CONFIGDIR=${CONFIGDIR} \
        JAVADOCLIMB=${JAVADOCLIMB}/$$i \
        JAVADOCTOP=../${JAVADOCTOP} \
        mktag
```

Thus JAVADOCTOP is always a relative path to the directory with *javadoc/,javafiles* and JAVADOCLIMB is the path from there to the sub...subdirectory currently being traversed in walking the subtree.

Once the *javadoc/,javafiles* is completed, including any recursive *makes*, the rule for *javadoc/index.html* can fire. It is in *imake.html* and reads as follows:

```
javadoc/index.html:: javadoc/,javafiles
        ${JAVADOC} -d javadoc `sort -n javadoc/,javafiles`
        ${RM} javadoc/,javafiles
```

This command sorts *javadoc/,javafiles* (numerically, in case any filename begins with digits) and passes the result as the list of files to ${JAVADOC}. The -d javadoc directs the result to the *javadoc* subdirectory. Lastly, the *,javafiles* list is deleted. The only remaining task is to arrange to install the newly generated documentation into the destination tree when Javadocs or GEE is made from the top-level *src/* directory. This is accomplished by the following rule in the top-level Amakefile:

```
GEE Javadocs:: install.time javadoc/index.html
        ${RM} -rf ${GEEHTMLDIR}/javadoc
        ${TAR} cf - ./javadoc | (cd ${GEEHTMLDIR}; tar xf -)
```

This command deletes a pre-existing *javadoc* subdirectory in the destination tree and copies the new one by creating a *tar* file of it and extracting from that *tar* file into the destination directory.

4.8 JAR Files

The Java developers have introduced the *jar* command to supersede the older and proprietary-to-someone-else *tar* and *zip* commands. In essence, *jar* compresses copies of a set of files and creates from them a *.jar* file. In most cases it is sufficient to collect all the *.class* files and put them in the *.jar* file. The GEE developers, however, wanted only a subset of the *.class* files, so it was necessary to create the *.jar* file from a list of files.

Problem: Some of the filenames to be included in the list contain dollar signs.

Solution: In the macro InstallFiles (Section 3.3.2), the list of names was generated with a wild-card expression:

```
package/ALLFILES.class
```

Since we wanted only a subset of this list, however, we needed an explicit list of names in the *Imakefile*. After some fumbling, we eventually discovered that using four dollar signs ($$$$) in names in the *Imakefile* was the only way to represent a single dollar sign in a filename!

Problem: The list of files on which the *.jar* file depends is not in the right form as an argument to the *jar* command.

Solution: The input list to *jar* needs to have the directory name separate from the filename. That is, files *a/b/t.class* and *a/b/u.class* have to be in the list as

```
    -C a/b/ t.class u.class
```

This problem was solved by writing a small C program, *gjardirs*, which does appropriate string processing to render the list of files into the list demanded by *jar*.

Given the above solutions, the rule for creating a *.jar* file is

```
#define JarFile(name, files)
compile:: name
name: files
      (filelist=`${ECHO} 'files'`; \
          ${JAR} cf name \
          `${CONFIGDIR}/gjardirs $$filelist`)
```

Here the `` `echo 'files'` `` trick protects against dollar signs, and the filelist is processed by *gjardirs* before being passed as an argument to *jar*.

5 Challenges from Other COTS Products

For many years, even non-COTS projects have used COTS compilers and libraries. The usual experience is that these COTS products remain stable throughout the life of the project or, at the least, that changes are upwardly compatible and do not affect the project. This experience does not hold in the current COTS environment; on average, a system with six COTS products having annual upgrade cycles will encounter a release every other month, and the majority of releases will be disruptive.

For GEE, the principal tool for dealing with new releases of a COTS product, say *xxx,* was to incorporate the reinstallation activity into a construction script in the *GEEBuild-Tree/external/*xxx/ directory. This script is also a tool for communicating to customer sites the activities that they must undergo to upgrade their COTS products to the version required by GEE. During the four months of initial GEE development, new releases were received for Java, JRun, and VisiBroker. All were readily incorporated by doing a *make* in the appropriate *external/*xxx/ directory.

Imakefiles are used in the *external/* tree and are constructed with *genmake*. This means that a new installation of GEE must first build and install the *src/config/* directory, then do appropriate system upgrades as described by the *external/* directories, and finally go back to build and install the rest of the *src/* directory.

Each COTS product added its own unique challenges to the construction/deployment task. In general, these were in the areas of

- installing the product
- adapting to new versions
- initiating servers that would respond to GEE
- establishing communication with GEE

5.1 Common Challenges

Enough problems occurred with two or more of the COTS products that it is useful to devote this section to them.

An early decision was whether to retain a copy of each COTS product release in the source archive. If the product is viable, its Web site will remain active, so the release could be re-fetched. However, in order to document our work fully and present a complete package to customers, we store the distribution binaries in the archive, usually as a single compressed, distribution-archived file. Ideally, of course, the *Imakefile* would contain instructions to re-fetch the installation from the Web site as needed, but the necessary tools and infrastructure are not yet in place.

Make rules in the *external/* subtrees were designed so the distribution would be unpacked and installed only once. Initially this scheme failed because timestamps on unpacked files reflect the time they were last changed before being packed, so they appear older than the archive and cause an unnecessary (and lengthy) repeat of the unpack/install step. This problem was resolved with a dummy file, *.unarchived*; it is created after unpacking the archive and triggers a new unpack operation only if the archive itself is replaced. Even this solution failed on one occasion when a new archive was inserted with a tool that set its date to its original date of creation instead of the date of insertion.

The *external/*/Imakefiles* contain some installation rules that define the Install make target and install files into the GEE directories. More commonly, files must be installed into directories owned by the system or devoted to a COTS product installation. Since the usual developer does not have privileges to write in these directories, an additional make target is provided called SystemInstall. (This target is generated from the SystemInstallFileAsNeeded macro, described in Section 3.3.2.) As part of installing a COTS product, the *make* command for this target must be executed by the root user. The implementation takes care to limit access, however, by ceding ownership of the installed file to the user who owns the destination directory, rather than root. Execution as root must usually be done by a system administrator, but there is no standard way to request such operations. Rather than rely on personal interaction or email, there could be a queue of actions, which the administrator would check and execute occasionally.

One of the steps that must be done as root is to install files that will initiate server processes whenever the operating system starts. Such servers are needed for Oracle, VisiBroker, JRun, and Netscape. On Solaris, a System V-like UNIX, these servers are started by files installed into */etc/rc.d*. For instance, the following file is installed from *external/vbroker/* so VisiBroker will be started at boot start time:

```
#!/bin/sh
VBROKER_HOME=GEE_VISIBROKER
VBROKER_OWNER=`/bin/ls -dl ${VBROKER_HOME}/adm/impl_dir \
     | awk '/impl_dir/{print $3}'`
/bin/su - ${VBROKER_OWNER} -c ${VBROKER_HOME}/bin/vbstartservers
```

Before installation, this script is expanded according to *gee.sed* to get the value of GEE_VISIBROKER, the location of the VisiBroker installation. The third line of the script determines which user owns the directory *./adm/impl_dir/* in the VisiBroker installation and then starts a shell as that user running the *vbstartservers* script. (The latter is also installed from *external/vbroker/*.)

Ideally, each developer has an independent testing environment and suffers no interference from tests by others. GEE supports this with a full sandbox build by allowing distinct instances of *gee.properties*, different Oracle databases, and variant Java class paths. The limitations of the evaluation version of JRun do not permit separate development, although these restrictions are lifted by JRun's Professional Edition. For a subset build, most of those distinctions were not possible; however, GEE does support separate CORBA servers by allocating a different Visibroker port to each developer.

5.2 Oracle Challenges

GEE uses an Oracle database for record storage and retrieval. Oracle is quite mature and no new releases occurred during GEE development. However, GEE relies on Oracle's Java Database Connectivity (JDBC) interface, which was not part of the previously installed Oracle. Since it is likely to be needed in any environment for GEE, the JDBC release is retained in *external/oracle/*. The *Imakefile* decompresses and expands it, but leaves its actual installation as a manual operation to be done in accordance with the distribution's *Readme* file.

Gee.properties has a number of entries for Oracle, such as the path to the installation, the database name, and the server host name and port. Properties exist for the Oracle "schema" name and password; these are changed for each GEE version. The gee.java.classpath property is augmented with the location of Oracle's Java library that comes as part of the JDBC distribution. (This library is labeled for use with JDK 1.1.1, but seems to work with JDK 1.2.)

For a full GEE system build, GEE's Oracle database must be created. Considerable experimentation was required to learn how to do this, and this experience has now been incorporated in the Imakefile. A small shell script, *runsql*, shown in Table 7, concatenates a set of Oracle command files and passes the result as a script to Oracle's *sqlplus* client tool. The action in the *Imakefile* is this:

```
    ${CSH} runsql dropdb dropdb createdb createfk
```

All these files, including the *runsql* script, are processed through *ExpandGeeProperties* to expand the GEE_... values.

```
#set variables required by sqlplus
setenv ORACLE_BASE GEE_ORACLE_BASE
setenv ORACLE_HOME GEE_ORACLE_HOME
setenv ORACLE_SID GEE_ORACLE_SID
setenv ORACLE_TERM GEE_ORACLE_TERM

#combine argument files into one temporary file
set tmpfile=/tmp/sqlscript.$$
rm -f $tmpfile
foreach f ($*)
        echo @$f >> $tmpfile
end

#run sqlplus
set PW=GEE_ORACLE_PASSWORD_FOR_SCHEMA
set DB=GEE_ORACLE_SCHEMA/${PW}
${ORACLE_HOME}/bin/sqlplus  ${DB} < $tmpfile
rm $tmpfile
```

Table 7: runsql.csh: Send Commands to Oracle

5.3 VisiBroker Challenges

VisiBroker is an implementation of the CORBA standard for "object request brokers." In CORBA, a process running a "service" implements an object with methods, a client program calls those methods, and the broker arranges the communication between the client and the service within its process.

The objects and methods offered by a service are described in a file written in IDL, an interface definition language. Visibroker, or any other CORBA implementation, provides a translator from IDL into a programming language, in GEE's case Java. The Java output from translating an IDL file is two-fold:

- a set of client objects having the defined methods
- a set of super-class files for objects implementing the services

Client programs make calls on the first, while semantics for the service are implemented by writing subclasses of the second.

The construction/deployment tasks for Visibroker include running the IDL translator, compiling the resulting Java files, installing all files, and registering the service so Visibroker knows how to start it. All these tasks are illustrated in Table 8, the *Imakefile* for the GEE business logic component, *GEEBuildTree/src/busLog/.*

```
IDLSource(Genotype, Genotype)
IDLModule(Genotype)
InstallIDLSource(Genotype.idl, ${GEELIBDIR}/idl)
InstallIDLModule(Genotype, ${GEECLASSESDIR})

EXTRACLASSES = ../gee:../rdbms:../security:../naming
JavaPackage(.)
InstallJavaPackage(., ${GEECLASSESDIR})
JavaPackage(Genotype/UserFactoryPackage)
InstallJavaPackage(Genotype/UserFactoryPackage, ${GEECLASSESDIR})

VERSION = GEE_VERSION
RegisterForActivation(IDL:gee.v${VERSION}/Genotype/UserFactory:1.0, \
        UserFactory,UserFactoryImpl)
```

Table 8: Imakefile *for a Component Describing a Service with IDL*

Note:

- *Genotype.idl* describes a CORBA service called LocService.
- The first two lines invoke the IDL transformation and the necessary Java compilation. (Note that there is no change to the Imakefile in order to switch to a different IDL converter.)
- The InstallIDLxxx lines copy the results of the IDL processing into the proper installation directories.
- Definitions nested within IDL interface declarations[10] cause extra packages to be created, which must also be compiled and installed. This is the purpose of the two lines for UserFactoryPackage.
- The last two lines register the new service, LocService, with Visibroker so it will be started when a request for that service arrives at the Visibroker port.
- VERSION is defined to avoid having vGEE_VERSION, which would not be converted by the C proprocessor.

An IDL source file may contain descriptions of a number of "modules," each of which compiles eventually into a Java package. Correspondingly, *gee.rls* provides one macro, IDLSource, for processing an IDL source file and another, IDLModule, for processing each module. It would have been preferable to have IDLSource describe all this processing; the macro was created with this in mind and its parameters include a list of the modules. However, the *imake* language is too weak to create the module processing rules this way, so the IDLModule macro is provided as well. In practice, the internals of the latter rule are identical to those for JavaPackage (they use a common sub-macro), but future developments could provide additional semantics for IDLModule.

[10] In *Genotype.idl*, the declaration of interface UserFactory contains
typedef sequence<octet> Credentials;

Ideally, IDLSource would compare the dates of the *.idl* file and one of its outputs to reprocess the *.idl* file when it has changed. This cannot be done because there is no file that is always created by processing an IDL source file. Instead, a dummy file called .srcname.*stubbed* is created and tested. The body of the macro looks like this:

```
compile:: .srcname.stubbed
.srcname.stubbed: srcname.idl
        <set environment> $${VB}/bin/idl2java srcname.idl
        ${TOUCH} .srcname.stubbed
```

The <set environment> code defines VB to be the Visibroker directory, defines other environment variables for VisiBroker, and defines Java variables so Java version 1.1 will be used. Idl2java is the only piece of GEE that needs Java 1.1, but it requires almost as much machinery as Java 1.2 in *gee.properties*, *gee.rls*, and *imake.tmpl*.

As noted above, IDL translation produces a set of Java superclasses for which subclass objects must be written. These subclasses must be accommodated in the *Imakefile* by adding rules for Java compilation and installation of these files. In Table 8, this is done with the usual rules for the default package as shown in lines 5-7.

Installed services are useless until processes executing them are started on the VisiBroker host. Every service could be started at system boot time, but the result could be many unused processes occupying system resources. Instead, VisiBroker has an object[11] activation daemon process—called *"oad"*—which listens for service requests and locates or starts a process for each. In order for *oad* to be able to activate a service, that service must first be registered. The following describes the Visibroker registration process from the standpoint of a user who has actually done it.

Each IDL module declaration may describe one or more named "interfaces," some of which correspond to services that can be registered and activated. Registration is done in GEE *Imakefiles* with the RegisterForActivation macro, which has three arguments:

1. the name the service will have in the repository
2. the name of the interface that the service advertises
3. the class name of the class to execute for that interface

(The documentation describes the first as the "repository name." This, it turns out, is not the name of a repository, but is a name that the service will have in the repository. The repository itself does not seem to have a name.)

A service's name in the repository has this form:

```
IDL:gee.vnn/modulename/interfacename/:1.0
```

[11] Technically, the service class is instantiated as an object, hence the name.

where *nn* is the value of gee.version, *modulename* is the name of the module, and *interfacename* is the name of the interface. (The ":1.0" is supposed to be a version number, but VisiBroker provides no support for any other value.) In GEE (version 01), the Genotype module contains a UserFactory interface, which is implemented in the UserFactoryImpl class; these names are the second and third parameters to RegisterForActivation. After determining the parameters, there is one more problem to solve before defining that macro.

Problem: Our first experiments with JDK1.2 and VisiBroker failed. Eventually we discovered that Java itself supports an object request broker and fields the calls on the CORBA application programmer interface; VisiBroker never sees them. Moreover, the JDK 1.2 class files for its broker are *not* on its CLASSPATH; no possible change to that path can put VisiBroker ahead of the Java broker. Instead, JDK 1.2 has a "bootclasspath" value that lists internal classes that are always loaded.

Solution: It was necessary for us to provide a value for the bootclasspath and list VisiBroker's class files ahead of those for JDK 1.2. Thus, the Java section of *gee.properties* became a bit more complex. Considerable time was spent in determining what values were needed on the bootclasspath, what their order should be, and what classes should be on the classpath.

Problem: VisiBroker can register services that are written in Java, but it provides no way to set the bootclasspath.

Solution: Use the syntax for registering a service written in C, but instead register the *gj* command and specify parameters so that it runs the correct Java program. The resulting registration macro is shown below:

```
#define RegisterForActivation(repname,interface,classnm)
install.time:: classnm.class
      GEE_VISIBROKER/bin/oadutil reg -r 'repname'\
          -o interface -cpp ${GEEBINDIR}/gj \
          -a `echo classnm | ${TR} '/' '.'` \
          -host GEE_VISIBROKER_HOST
```

Problem: There is no rule for what RegisterForActivation calls classnm.class, which is supposed to be the result of compiling the implementation files. If the file does not exist, *make* fails because it cannot figure out what to do.

Solution: None. If the failure occurs, *make* must be rerun with the compile target, which will rebuild all classes. (The rebuild will occur because of the file absence; this is one advantage of the *needjc* approach over a dummy file approach.)

Since multiple developers may be implementing and testing new services, it is valuable to be able to distinguish between them. We were able to do so because VisiBroker allows multiple instances of *oad* to be running, each listening on a different port. For GEE we wrote a *vbstart* command that starts a set of VisiBroker processes listening on a port named as a command argument. By setting an environment variable (VBPORT), a developer arranges that services registered will be available on that port and that any client programs that are started will use that same *oad*. As with *gj*, *vbstart* incorporates the system environment information from *gee.properties*.

Debugging is further supported by VisiBroker with a repository of IDL files, such as are installed in *GEELIBDIR/idl/* by InstallIDLSource. The best documentation of this repository is GeeBuild.doc, the user's guide to the GEE construction scripts [Hansen 98a].

5.4 JRun and the Netscape Web Server

One of three modes of user interface for GEE is via "servlets." A servlet is a Java package associated with a URL and a Web server; when that URL is requested from the server, the servlet is executed and its output is returned as the requested page. Java provides *jsdk*, the Java Servlet Development Kit, as an application programmer interface to servlets, but third-party software is needed to bridge the Web server to the Java execution environment. After some searching on the Web, the only product that connected servlets written in JDK 1.2 was JRun from Live Software. We downloaded the evaluation distribution and incorporated it into GEE in the *external/livesoftware/* directory.

As part of setting up JRun, the Web server (in our case Netscape Enterprise Server) must be told to communicate with JRun for selected incoming URLs. This process is initiated by the *external/livesoftware/Imakefile* after installing JRun. However, the Netscape server must then be manually stopped and restarted to incorporate the changes into its environment. Moreover, the JRun server process must be started before restarting Netscape and at every system reboot thereafter. The *Imakefile* does not immediately start the JRun server, but it does install a script to do so and arranges for the script to be called on subsequent reboots.

JRun does not start a new Java execution for each incoming servlet URL because doing so would incur the response time penalty of Java startup. Instead, JRun loads the class referenced in the URL into its own process and executes it there.

Problem: To execute classes, they must be on Jrun's CLASSPATH; The JRun convention is for them to be in a directory within the JRun installation. However, to put them there would further complicate the GEE installation process by requiring permission to modify the JRun directory.

Solution: A single GEE destination directory name—*../gee/jrunclasses/*—is added to the JRun CLASSPATH. This is done from the *external/livesoftware/Imakefile* by a *sed* script that modifies a property file within the JRun installation tree. To provide for redirection of servlet calls from one GEE version to another, the *jrunclasses* name is in fact a symbolic link to the *classes* subdirectory of the GEE installation tree for one of the several extant versions of GEE.

The *jrunclasses/* solution has an appealing simplicity but also a few drawbacks. One is that the servlet class files are in the same directory with other class files. Consequently, other class files may conceivably be initiated from a URL. Since these classes do not implement jsdk, the worst that will happen is a confusing error report to the user. Such reports should be replaced with more user-friendly responses in a professional installation. A second drawback is that if a servlet has been initiated, it is necessary to stop and restart JRun to use a new version of that servlet. A last drawback is that one environment must be used for both testing and production use. All these drawbacks are avoided by the "Professional" version of JRun, but the required investment has not been justified by GEE's modest needs.

6 Miscellaneous Issues

In this chapter, we cover a few additional topics that posed significant problems for construction and deployment.

6.1 Security

Since security issues are important to all Software Engineering Institute customers, security issues were a major concerns of GEE development. We strove to incorporate state-of-the-art certificate technology, but we were stymied by this technology's immaturity.

Problem: Passwords for Oracle and user certificate databases are contained unencrypted in *gee.properties*.

Solution: No solution was implemented. Encrypting passwords in *gee.properties* would not help much because an interloper could decrypt them with whatever process GEE itself uses. Ideally, a password would be entered manually for every system restart. However, if hands-free automatic restarts are desirable, the passwords must be in a file somewhere on the system. *Gee.properties* could give the location of this file, to which access would be limited by security measures outside GEE.

Problem: Where should the user certificate database be stored?

Solution: No good solution was developed:

In principle, a certificate infrastructure manages a user's certificates. An authority issues certificates that identify the user and are tied to permissions for various GEE actions. An encrypted form of these certificates is stored in the user's certificate database. When a user begins an interactive session, the certificate is fetched from the database (or from a ring or other wearable device) and decrypted according to some password entered by the user. Once the user has a certificate, it is passed to each application where it specifies the user's rights.

No infrastructure for certificates existed on the GEE deployment systems, so the certificate database was made part of the GEE distribution, and we attempted to perform certificate processing within the GEE client application. To mimic the eventual certificate environment, certificates are stored on the client host rather than centrally stored at the GEE server. The question then arises: In what directory on the client's host should the certificate database be stored? Since clients were to run on both Windows and UNIX platforms, a full solution to this problem would have required developing deployment packages for both these environments. Instead, we decided to store the database in the directory that was "current" at the time the client was launched. This is a simple concept for UNIX platforms, but it required considerable experimentation on Windows to find out which was the current directory. In either case, the certificate database is manually copied to the user environment.

6.2 Cron: Daily Processing

In *external/,* alongside the directories for COTS products, there is a directory called *external/cron/.* It describes GEE processing that is performed daily, as dictated by a shell script that is initiated daily at 3:07 a.m. by the UNIX *cron* mechanism. This script ensures that the installed GEE is up-to-date by deleting the entire installation, refetching the sources from the archive, and executing a complete rebuild. The script also copies the GEE source archive to a host where files are backed up daily. Originally, these backups were saved for a week, but when the disk overflowed this was changed to saving them for only two days.

Problem: If there is any failure during the build, no executable GEE is installed.

Solution: Constant vigilance is required to keep updating files so the rebuild succeeds. Fortunately, there is no actual usage of GEE, so failures are not catastrophic. Ideally, the daily build would be augmented to send an email message if it failed.

A third task of the daily processing is for JRun. JRun can access only one library directory, so it is told about the name *gee/jrunclasses/.* The daily *cron* process makes this name be a symbolic link to the classes library for the most current stable version of GEE.

6.3 Beginning a New GEE Version

From time to time, the current version of GEE is deemed "finished," and work begins on a new version. Usually a finished version is working, but seldom in all facets; the impetus for change is to attempt new approaches requiring major changes not only to the files, but also in the number of files or the directory structure. To retain the old version, the entire source archive is copied and stored along with an installation directory resulting from that version.

Changing to a new version is begun by modifying *gee.properties*, especially gee.version and the name of the Oracle database schema. Other properties and source files are then changed as desired. The most difficult manual steps are to update the Web pages. This process has not been mechanized because of the number of differences between what should be offered by each version. So far, it has been possible to provide intranet Web site access to executable instances of old versions; often, however, these begin to fail as resources are redirected to the new version.

7 Conclusion

Our initial goal was to develop a construction/deployment system for GEE that would implement the four build scenarios in Figure 1: build-install, subset build, sandbox build, and customer build. The necessary flexibility was introduced by dividing the sources into component subdirectories, by centralizing definitions in *gee.properties*, and by using *imake* to generate *Makefiles* from platform and site-dependent definitions files.

The various COTS products incorporated in GEE had a considerable influence on the construction/deployment system. Java required considerable ingenuity to adapt the make processor to suit its requirements. Together with Java, the Visibroker, Oracle, and JRun products added complexities in the areas of installation, version change, server initiation, and establishing communication with GEE processes.

We began this report with two principles:

Repeat not.
Delay binding.

Although the text has not explicitly invoked these principles, they have been influential throughout:

- The *gee.properties* file is the central source for many of the values that should not be repeated elsewhere. The properties mechanism permits using names instead of the values themselves, and it avoids repetition of the values so they can be revised in a single place.
- *Makefiles* typically have considerable repetition both within files and from one file to another. This repetition is avoided by using *imake* to macro-generate a *Makefile* from an *Imakefile*, the template *imake.tmpl*, and the macro definitions in *gee.rls*. The rules in *gee.rls* generate lengthy pieces of code for each invocation of a macro. Not only are the *Imakefiles* much smaller than *Makefiles*, but porting GEE to another platform can be done primarily by changing these definition files and not other files throughout the system.
- Typing commands can be repetitious. In GEE this is avoided by having a number of ad hoc commands implemented as shell scripts: *gj* for Java execution, *vbstart* for initiating a VisiBroker service, and so on. These scripts incorporate knowledge of the environment from gee.properties and make it available to avoid repeated typing of options and the consequent possibility of errors.
- When releases of COTS products arrive, the processing to incorporate them is much the same as for the previous release. All this processing is captured in the *external/*/Imakefiles,* so typing in these commands need not be repeated.

The principle of delayed binding is observable in GEE, less in its presence than in the problems caused by its absence. The little shell scripts are preprocessed with *gee.sed* and do not change when *gee.properties* changes. Thus, they need to be reinstalled to get a consistent environment. Similar problems exist because file names, especially in HTML files, are fixed via preprocessing with *gee.sed*; consequently, it is not possible to move the installed files without rebuilding the system.

In a future effort, we would explore further measures to defer binding. If beginning a longer term effort, it might be fruitful to explore an alternative to *imake/make* that provided more capabilities and fewer syntactic distractions.

References

[Andrew 96]	Andrew Consortium. *The Andrew Consortium*. Available WWW <URL: http://www.cs.cmu.edu/~AUIS/> (1996).
[Dubois 96]	Dubois, Paul. *Software Portability with Imake, 2nd Edition*. Sebastopol, CA: O'Reilly & Associates. Available WWW <URL: http://www.primate.wisc.edu/software/imake-book/index.html> (1996).
[Feldman 86]	Feldman, S. I. "Make - a program for maintaining computer programs." *UNIX Programmer's Supplementary Documents* Vol. 1 (PS1), USENIX Assn., 1986 (pp. PS1:12-1–PS1:12-9).
[Fulton 89]	Fulton, Jim. *Configuration Management in the X Window System*. Cambridge, MA: X Consortium, MIT Laboratory for Computer Science, pp. 12. Available WWW <URL: http://www.primate.wisc.edu/software/imake-stuff/fulton.txt> (1989).
[Hall 97]	Hall, Richard S.; Heimbigner, Dennis; and Wolf, Alexander L. *Software Deployment Languages and Schema* (CU-SERL-203-97). Dept. of Computer Science, University of Colorado, December 1997. Available WWW <URL: http://www.cs.colorado.edu/users/rickhall/deployment/SchemaPaper/Schema.html> (1997).
[Hansen 98a]	Hansen, W. J. *Gee Building*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, December 1998. (Available within [Hansen 98b] as geebuild/doc/GeeBuild.doc.)
[Hansen 98b]	Hansen, W. J. GEE construction/deployment scripts sources. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. Available WWW <URL: http://www.sei.cmu.edu/staff/wjh/geebuild.tar> (1998).

[Hansen 99] Hansen, W. J. “Deployment Descriptions in a World of COTS and Open Source,” *Ninth International Symposium on System Configuration Management* (SCM-9). Toulouse, France, Sept. 1999. Heidelberg: Springer Verlag LNCS, 1999. Available WWW <URL: http://www.sei.cmu.edu/staff/wjh/DeployDesc.html>.

[Inprise 99] Inprise. *CORBA Technology from Inprise*. Scott’s Valley, CA. Available WWW <URL: http://www.inprise.com/visibroker/> (1999).

[Live Software 99] Live Software. *Live Software Products: JRun*. Cupertino, CA. Available WWW <URL: http://www.allaire.com/products/jrun> (1999).

[MacKenzie 98] MacKenzie, David and Elliston, Ben. *Autoconf - Creating Automatic Configuration Scripts, Edition 2.13*. Cambridge, MA: Free Software Foundation. Available WWW <URL: http://www.gnu.org/manual/autoconf/index.html> (1998).

[Netscape 99] Netscape Communications. *Netscape Products*. Mountain View, CA. Available WWW <URL: http://home.netscape.com/download/index.html. (1999).

[Oracle 99] Oracle. *Oracle8 - Database Servers – Products*. Redwood Shores, CA. Available WWW <URL: http://www.oracle.com/database/oracle8/> (1999).

[Sun 99a] Sun Microsystems. *Java™ Technology Home Page*. Palo Alto, CA. Available WWW <URL: http://www.java.sun.com> (1999).

[Sun 99b] Sun Microsystems. *Code Conventions for the Java™ Programming Language*. Palo Alto, CA. Available WWW <URL: http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html> (1999).

[Tilbrook 96] Tibrook [sic], D. “An Architecture for a Construction System,” 76-87. *Software Configuration Management, ECOOP’96 SCM-6 Workshop*. Berlin, Germany, March 1996. Springer, Lecture Notes in Computer Science # 1167. Available WWW <URL: http://www.cleanscape.net/stdprod/qef/qefwhite.html> (1996).

[Wallnau 98] Wallnau, K.; Hansen, W. J.; Hissam, S.; Long, F.; and Seacord, R. *GEE Vee Zero Point One*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, August 1998. (Available within [Hansen 98b] as geebuild/doc/GeeVeeZeroPointOne.doc.)

Appendix A: Properties File: gee.properties

At runtime, GEE components that are written in Java retrieve property values from *gee.properties*, below, via the GeeProperties class. Other GEE components get these values at build time via processing with *gee.h* or *gee.sed* as generated from *gee.properties* by *Defines.java*. (See Appendix B for *Defines.java*.)

```
# Properties file for GEE
#
# This file is installed in gee.destination.directory/config/gee.properties
# and are preprocessed to create gee.h and gee.sed in the same directory.
#
# WARNING
# WARNING No property name should be a substring of any other property name.
# WARNING
#



#________________________________________
# GEE
#    the root of installation tree
gee.destination.directory = /usr/local/gee/dest

#    version number. The value must be only letters and digits.
gee.version = 03

#    host for gee web site
gee.web.host = gc.sei.cmu.edu

#________________________________________
# WEB SITE
# For the descriptions, ".../gee" means the value of gee.destination.directory

# The overall gee website
# ( This is established in the webserver by mapping GEE to .../gee
gee.website.prefix = http://${gee.web.host}/GEE

# the URL to refer to html files. The html files themselves must go
# into the directory to which the server maps this URL.
gee.html.prefix = http://${gee.web.host}/GEE/dest/html

# The url location for servlets
# ( The web browser maps /servlet/ to the jrun style.
# ( The jrun style sends the URL to jrun.
# ( jrun maps /servlet/ so the next word is a class name
# ( (see jrun/jsm-default/services/jse/properties/rules.properties)
# ( The class is sought in jrun.servletdir
# ( which is set in jrun/jsm-defaults/services/jse/properties/jrun.properties
# ( to the directory /usr/local/gee/jrunclasses,
# ( which is a symlink to one of
# (   gee/dest/classes, gee/dest01/classes, gee/dest02/classes, ...
gee.servlet.prefix = http://${gee.web.host}/servlet

# The url location for cgi-bin executables
# ( this is mapped by the web server to .../gee/cgi-bin
gee.cgi.prefix = http://${gee.web.host}/GEE/cgi-bin

#________________________________
# SECURITY

#   location of key store (per client)
gee.keystore = ${gee.destination.directory}/etc/oscStore-v0.3

gee.password.for.keystore = 08j06a96s


#________________________________
# VISIBROKER

#   the location of Visibroker.
gee.visibroker = /usr/local/vbroker
gee.visibroker.port = 14000
gee.visibroker.host = gc.sei.cmu.edu


#________________________________
# ORACLE
#   the location of Oracle.
gee.oracle.base = /u01/app/oracle

#   the particular Oracle version in use.
```

```
gee.oracle.home = ${gee.oracle.base}/product/8.0.3

#   the userid of the owner of gee.oracle.home
gee.oracle.owner = oracle

#   the oracle data base name
gee.oracle.sid = test

#   the oracle terminal type
gee.oracle.term = xsun5

#   the schema used for GEE
gee.oracle.schema = TARZAN
gee.oracle.password.for.schema = JANE

#   oracle host an port
gee.oracle.server = @gc
gee.oracle.port = 1521


#______________________________________
#JAVA

# The Java of choice is jdk1.2.
#   location of java jdk1.2; binaries are in gee.java/bin
gee.java.home = /usr/local/jdk1.2

#   path list (colon-separated) of directories containing Java class
#   files. Also among the entries can be classes.zip files.
gee.java.bootclasspath =\
    ${gee.visibroker}/lib/vbjorb.jar:\
    ${gee.visibroker}/lib/vbjapp.jar:\
    ${gee.java.home}/jre/lib/i18n.jar:\
    ${gee.java.home}/jre/lib/rt.jar

gee.java.classpath =\
    ${gee.destination.directory}/classes:\
    ${gee.oracle.base}/product/8.0.3/jdbc/lib/classes111.zip:\
    ${gee.java.home}/lib/jce12-dom.jar:\
    ${gee.java.home}/lib/jsdk.jar:\
    ${gee.java.home}/lib/ldapjdk.jar:\
    ${gee.java.home}/lib/capsapi_classes.zip:\
    ${gee.java.home}/lib/classes.zip
```

```
#We also (grudgingly) support compilation and execution with jdk 1.1
gee.java11.home = /usr/local/jdk1.1
gee.java11.classpath =\
    ${gee.destination.directory}/classes:\
    ${gee.visibroker}/lib/vbjorb.jar:\
    ${gee.visibroker}/lib/vbjapp.jar:\
    ${gee.oracle.base}/product/8.0.3/jdbc/lib/classes111.zip:\
    ${gee.java11.home}/lib/jce12-dom.jar:\
    ${gee.java11.home}/lib/ldapjdk.jar:\
    ${gee.java11.home}/lib/capsapi_classes.jar:\
    ${gee.java11.home}/lib/classes.zip

#___________________________________
# JRun from Live Software, Inc.

gee.jrun.home = /usr/local/jrun
```

Appendix B: Properties File Converter: Defines.java

Defines.java, the program below, processes *gee.properties* to produce *gee.h* and *gee.sed*. (See Appendix A for the *gee.properties* file.)

```
/** Defines.java

    This program converts a Java properties file into a
    .h file with #defines file suitable for the C preprocessor.
    It also produces a .sed file with s/.../.../g commands
    for the same expansions.

    Lines beginning with # are deleted.
    Other lines are in the form

        property.name = value string extending to end-of-line

    These lines are converted for the .h file as
        #define PROPERTY_NAME value string extending to end-of-line
    Note that the property name is converted to all caps and
    periods are converted to underline.
    The conversion for the .sed file is
        s/PROPERTY_NAME/value string extending to end-of-line/g

    The first argument is the path to the properties files,
    ending in ".properties"  The output files are the same but
    have the extension .h and .sed.

    After reading in the .properties file with java.util.Properties.load(),
    the property values are scanned for subsequences ${...}. If the
    contents name a property, the value of that property is substituted
    for the entire subsequence. Prior to the substitution, the value
    is itself scanned for possible substitutions by this rule.
    Recursion is detected by limiting this process to ten levels.
    The sequence $${ is converted to ${
*/
```

```
import java.io.BufferedWriter;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Properties;


public class Defines {

    static java.util.Properties props = new java.util.Properties();


// write a single line to a BufferedWriter
private static void writeLine(java.io.BufferedWriter f, String s)
    throws java.io.IOException {
    f.write(s, 0, s.length());
    f.newLine();
}


// choose a delimiter which is not among the characters of a
private static char chooseDelim(String a) {
    String candidates =
        "\":;.,/^%$#@!~‘><][]{|+=-)(*&?"
        + "1234567890qwertyuiopasdfghjklzxcvbnm"
        + "QWERTYUIOPLKJHGFDSAZXCVBNM_";

    while (candidates.length() > 0) {
        char thisChar = candidates.charAt(0);
        if (a.indexOf(thisChar) < 0)
            return thisChar;
        candidates = candidates.substring(1);
    }

    throw new IllegalArgumentException(
        "no character available as delimiter");

}


// in val, expand ${...} to value of ...
// cnt increments for expansion depth and does not exceed 10
//   (cnt does NOT increment when expanding a tail of a string)
// the algorithm is highly inefficient and recursive,
//   not too many ${...} are expected
private static String expand(String val, int cnt) {
    if (cnt >= 10) return val;   // avoid recursion
    int dolloc = val.indexOf("${");                                   //
location of first "${"
    if (dolloc < 0) return val;   // no "${"
    if (dolloc > 0 && val.charAt(dolloc-1) == '$')
        // "$${" -> "${"
        return val.substring(0,dolloc-1) + "$"
            + expand(val.substring(dolloc+1), cnt);
    int rightloc = val.indexOf('}', dolloc);
    if (rightloc < 0) return val; // no trailing '}'

    // expand ${...}
    String name = val.substring(dolloc+2, rightloc);
    return val.substring(0, dolloc)
        + expand((String)props.get(name), cnt+1)
        + expand(val.substring(rightloc+1), cnt);
}

public static void main(String args[]) throws java.io.IOException {

    // setup filenames and file objects
    //
    String inFileName = args[0];
    String hFileName, sedFileName;
    if ( ! inFileName.endsWith(".properties"))
        throw new IllegalArgumentException(
                "file name must ends with \".properties\"");
    hFileName = inFileName.substring(0,
            inFileName.length()-".properties".length()) + ".h";
    sedFileName = inFileName.substring(0,
            inFileName.length()-".properties".length()) + ".sed";
    System.out.println(inFileName + " => " + hFileName
            + " & " + sedFileName);

    java.io.FileInputStream inFile
            = new java.io.FileInputStream(inFileName);
    java.io.BufferedWriter hFile
            = new java.io.BufferedWriter(
                new java.io.FileWriter(hFileName));
    java.io.BufferedWriter sedFile
            = new java.io.BufferedWriter(
                new java.io.FileWriter(sedFileName));

    props.load(inFile);
    inFile.close();
    java.util.Enumeration pe = props.keys();

    while (pe.hasMoreElements()) {
        String key = (String)pe.nextElement();
        String keyOut = key.replace('.', '_').toUpperCase();
        String value = expand((String)props.get(key), 0);
        char delim = chooseDelim(keyOut+value);

        writeLine(hFile, "#define " + keyOut + " " + value);
        writeLine(sedFile, "s" + delim + keyOut
                    + delim + value + delim + "g");
    }

    hFile.close();
    sedFile.close();
}

}
```

Appendix C: Macros for Imake: gee.rls

The *imake* processor expands macro calls in the *Imakefile* by using the macros defined in *gee.rls*, below. The string "@ @\" at the end of a line is converted into a new line in the expansion of the macro. If there is a "\" before that, it is left in the output and is eventually passed along to the shell where it signals that the line break should be ignored.

```
/* gee.rls - macro (rules) for Imakefiles

    Copyright Carnegie Mellon University 1998

$Disclaimer:  $

$Hdr: /home/wjh/gee/src/config/gee.rls 1.3 $
*/


#define NOSHERRORS case '$(MFLAGS)' in *[ik]*) set +e;; esac
#define SHVERBOSE set -x

#define ALLFILES *

/*=================================================*\
 * COMPILATION RULES
\*=================================================*/

/* IDLSource(srcname, modulenames)
    Executes idl2java if srcname.idl has changed.
    Will create a directory and contents for each module named
    in srcname.idl. The list modulenames is used for the
    'clean' target;  all named directories are deleted in full.
    For each module, there should be an instance of IDLModule
    and an InstallIDLModule.
    For each Object which is a separate process,
    the Imakefile must have a RegisterVBServer.
    The Imakefile should also have
        JavaPackage(.)
        InstallJavaPackage(.)
    so it can deal with the xxxImpl.java files
```

```
*/
#define IDLSource(srcname,modulenames)                                        @@\
MACROINVOKED = _IDLSource(srcname,modulenames)                                @@\
COMMA = ,                                                                     @@\
Sedify(srcname.idl, $(CONFIGDIR)/gee.sed, ${COMMA}srcname.idl)                @@\
compile:: .srcname.stubbed                                                    @@\
.srcname.stubbed: ${COMMA}srcname.idl                                         @@\
    JAVA_HOME=GEE_JAVA11_HOME; export JAVA_HOME; \                            @@\
    ALT_JDK=$${JAVA_HOME}; export ALT_JDK; \                                  @@\
    VB=GEE_VISIBROKER; VBROKER_ADM=$${VB}/adm; \                              @@\
    LD_LIBRARY_PATH=$${VB}/lib:$${LD_LIBRARY_PATH}; \                         @@\
    PATH=$${JAVA_HOME}/bin:$${VB}/bin:$${PATH}; \                             @@\
    CLASSPATH=${CLASSES11}:GEE_JAVA11_CLASSPATH;\                             @@\
    export VBROKER_ADM; export LD_LIBRARY_PATH; \                             @@\
    export PATH; export CLASSPATH; \                                          @@\
    $${VB}/bin/idl2java ${COMMA}srcname.idl                                   @@\
    $(TOUCH) .srcname.stubbed                                                 @@\
clean::                                                                       @@\
    $(RM) -rf modulenames

/* IDLModule(modulename)
    Arranges to compile the java files in the named module,
    which must also be the name of a directory.
*/
#define IDLModule(modulename)                                                 @@\
MACROINVOKED = _IDLModule(modulename)                                         @@\
JavaPackageInternal(modulename,JAVACOMPILE)


/* RMIStubs(classname)
    Takes an RMI class and runs rmic on it to generate
    stub and skeleton classes
    ??? how to generate dependency for classname.class???
*/
#define RMIStubs(classname)                                                   @@\
MACROINVOKED = _RMIStubs(classname)                                           @@\
EMPTY =                                                                       @@\
compile:: Concat(classname,_Stub.class)                                       @@\
Concat(classname,_Stub.class): classname.class                                @@\
    $(RMIC) classname                                                         @@\
install:: Concat(classname,_Stub.class)                                       @@\
clean::                                                                       @@\
    $(RM) -rf Concat(classname,_Stub.class) Concat(classname,_Skel.class)
```

```
/* WARNING: There must be NO SPACES around the first argument */
#define JavaPackageInternal(pkgname, compiler)                        @@\
compile::                                                             @@\
    @(NOSHERRORS; \                                                   @@\
    jfiles=`${CONFIGDIR}/needjc pkgname/ALLFILES.java`; \             @@\
    if [ "X$${jfiles}X" != "XX" ] ; then \                            @@\
        $(ECHO) compiler $${jfiles}; \                                @@\
        $(compiler) $${jfiles}; \                                     @@\
    fi; exit 0)                                                       @@\
clean::                                                               @@\
    $(RM) pkgname/ALLFILES.class                                      @@\
javalist::                                                            @@\
    (cd $(JAVADOCTOP); \                                              @@\
        $(LS) -1 $(JAVADOCLIMB)/pkgname/ALLFILES.java) \              @@\
            >> $(JAVADOCTOP)/javadoc/,javafiles

/* JavaPackage(packagename)
    The packagename should be given with '/' instead of '.'
    The .java files for the package should be in the
    subdirectory ./packagename.
    The .class files will be built in the same directory.
    Any time any .java file in the package is modified,
    all .java files will be recompiled.
*/
#define JavaPackage(pkgname)                                          @@\
MACROINVOKED = _JavaPackage(pkgname)                                  @@\
JavaPackageInternal(pkgname,JAVACOMPILE)

/* Java11Package(packageName) same as JavaPackage, but uses jdk1.1 */
#define Java11Package(pkgname)                                        @@\
MACROINVOKED = _Java11Package(pkgname)                                @@\
JavaPackageInternal(pkgname,JAVA11COMPILE)


/* JarFile(name, files)
    Creates a .jar file containing the named files
    and a default manifest. The 'name' must end with the
    .jar extension.
    Each dollar sign in a file names must be represented
    with four (4) consecutive dollar-signs. (!)
*/
```

```
#define JarFile(name, files)                                        @@\
MACROINVOKED = _JarFile(name, files)                                @@\
compile:: name                                                      @@\
name: files                                                         @@\
    (filelist=`$(ECHO) 'files'`; \                                  @@\
        $(JAR) cf name `${CONFIGDIR}/gjardirs $$filelist`)

/* JarFileWithManifest(name, files)
    Creates a .jar file containing the named files
    and manifest META-INF/MANIFEST.MF. The 'name' must
    end with the .jar extension. The manifest must exist
    with the name MANIFEST.MF in a subdirectory named META-INF.
    Each dollar sign in a file names must be represented
    with four (4) consecutive dollar-signs. (!)
*/
#define JarFileWithManifest(name, files)                            @@\
MACROINVOKED = _JarFileWithManifest(name, files)                    @@\
compile:: name                                                      @@\
name: files                                                         @@\
    (filelist=`$(ECHO) 'files'`; \                                  @@\
        $(JAR) cmf META-INF/MANIFEST.MF name \                      @@\
            `${CONFIGDIR}/gjardirs $$filelist`)


#define Sedify(infile, script, outfile)                             @@\
compile:: outfile                                                   @@\
outfile: infile                                                     @@\
    $(SED) -f script infile > outfile                               @@\
clean::                                                             @@\
    $(RM) outfile


/* ExpandGeeProperties(infile, outfile)
    Infile is copied to outfile, except that the variables defined
    in gee.properties are replaced with their values.
    (sed is used with config/gee.sed as the script.)
*/
#define ExpandGeeProperties(infile, outfile)                        @@\
MACROINVOKED = _ExpandGeeProperties(infile, outfile)                @@\
Sedify(infile, $(CONFIGDIR)/gee.sed, outfile)
```

```
/* ****************************************************************** *\
 * INSTALLATION RULES

recommended rules:
    InstallJavaClass  InstallDocs  InstallShScript  InstallCshScript
    InstallDataFiles

\* ****************************************************************** */

/* these rules could be rewritten to have the destination file dependant on
    the source file instead of the artificial 'install.time'.
*/


#define InstallFiles(files, mode, dest)                              @@\
install.time:: files                                                 @@\
    @(NOSHERRORS; \                                                  @@\
    for i in `$(ECHO) '$?'`; do \                                    @@\
        (SHVERBOSE; $(INSTALL) mode $$i dest/$$i); done)
    /* the weird `echo '$?'` manages to protect
    against file names containing $ */


#define InstallJavaPackage(package,dir)                              @@\
MACROINVOKED = _InstallJavaPackage(package, dir)                     @@\
MkdirTarget(dir/package)                                             @@\
InstallFiles(package/ALLFILES.class, $(INSTCLASSFLAGS), dir)

#define InstallIDLSource(idlfile,dir)                                @@\
MACROINVOKED = _InstallIDLSource(idlfile,dir)                        @@\
InstallFiles(idlfile, ${INSTDATAFLAGS}, dir)


#define InstallIDLModule(package,dir)                                @@\
MACROINVOKED = _InstallIDLModule(package, dir)                       @@\
MkdirTarget(dir/package)                                             @@\
InstallFiles(package/ALLFILES.class, $(INSTCLASSFLAGS), dir)

#define InstallJarFile(file,dir)                                     @@\
MACROINVOKED = _InstallJarFile(file,dir)                             @@\
InstallFiles(file,$(INSTCLASSFLAGS),dir)
```

```
#define InstallDocs(files, dest)                                          @@\
MACROINVOKED = _Install(files, dest)                                      @@\
InstallFiles(files, $(INSTDOCFLAGS), dest)

#define InstallDataFiles(files, dest)                                     @@\
MACROINVOKED = _InstallDataFiles(files, dest)                             @@\
InstallFiles(files, $(INSTDATAFLAGS), dest)

#define InstallLibFiles(files, dest)                                      @@\
MACROINVOKED = _InstallLibFiles(files, dest)                              @@\
InstallFiles(files, $(INSTLIBFLAGS), dest)


/* RegisterForActivation(repname,interface,classnm)
    Registers with the object activation daemon (oad)
    the indicated interface.
    repname - the repository name for the object.
        By convention the form is
            IDL:prefix/modulename/interfacename:1.0
        where for gee the prefix is gee.vNN,
        where NN is the value of gee.version
    interface - the name following the keyword interface
    classnm - the name of the java class
        which registers the interface by calling
        boa.obj_is_ready(impl), where impl is a
        java object of class interfaceImpl
        It is assumed that this class is in the default
        package.
    example:
    RegisterForActivation(IDL:gee.v02/Genotype/UserFactory:1.0,
            UserFactory,UserFactoryImpl)
*/
#define RegisterForActivation(repname,interface,classnm)                  @@\
MACROINVOKED = _RegisterForActivation(repname,interface,classnm)          @@\
install.time:: classnm.class                                              @@\
    -GEE_VISIBROKER/bin/oadutil unreg -r 'repname' \                      @@\
         -o interface -host GEE_VISIBROKER_HOST                           @@\
    GEE_VISIBROKER/bin/oadutil reg -r 'repname' \                         @@\
        -o interface -cpp ${GEEBINDIR}/gj -a classnm \                    @@\
        -host GEE_VISIBROKER_HOST


#ifdef SCOUNIX
```

```
#define InstallCshScript(file, dest)                                    @@\
MACROINVOKED = _InstallCshScript(file, dest)                            @@\
install.time:: file                                                     @@\
    $(RM) /tmp/,file                                                    @@\
    $(ECHO) \: "\n/bin/csh -f -s csh \$$* \                             @@\
        << '==EOF=='\nshift" > /tmp/,file                               @@\
    $(CAT) file >> /tmp/,file                                           @@\
    $(ECHO) "==EOF==" >> /tmp/,file                                     @@\
    $(INSTALL) $(INSTPROGFLAGS) /tmp/,file dest                         @@\
    $(RM) /tmp/,file
#define InstallShScript(file, dest)                                     @@\
MACROINVOKED = _InstallShScript(file, dest)                             @@\
install.time:: file                                                     @@\
    $(RM) /tmp/file                                                     @@\
    $(ECHO) \: > /tmp/,file                                             @@\
    $(CAT) file >> /tmp/,file                                           @@\
    $(INSTALL) $(INSTPROGFLAGS) /tmp/,file dest                         @@\
    $(RM) /tmp/,file
#else /* SCOUNIX */

#define InstallShScript(file, dest)                                     @@\
MACROINVOKED = _InstallShScript(file, dest)                             @@\
InstallFileToFile(file, $(INSTPROGFLAGS), dest)
#define InstallCshScript(file, dest)                                    @@\
MACROINVOKED = _InstallCshScript(file, dest)                            @@\
InstallFileToFile(file, $(INSTPROGFLAGS), dest)
#endif /* SCOUNIX */


/* deprecated rules */

/* IDLPackage(modulename)
    Executes idl2java if modulename.idl has changed.
    Incorporates all the rules for building the
    resulting class.
    The Imakefile must also have
        InstallIDLPackage(destination-directory, modulename)
    and for each Object which is a separate process,
    the Imakefile must have a RegisterVBServer.
    The Imakefile should also have
        JavaPackage(.)
        InstallJavaPackage(.)
    so it can deal with the xxxImpl.java files

*/
#define IDLPackage(modulename)                                          @@\
MACROINVOKED = _IDLPackage(modulename)                                  @@\
compile:: modulename/.modulename                                        @@\
modulename/.modulename: modulename.idl                                  @@\
    JAVA_HOME=GEE_JAVA11_HOME; export JAVA_HOME; \                      @@\
    ALT_JDK=$${JAVA_HOME}; export ALT_JDK; \                            @@\
    VB=GEE_VISIBROKER; VBROKER_ADM=$${VB}/adm; \                        @@\
    LD_LIBRARY_PATH=$${VB}/lib:$${LD_LIBRARY_PATH}; \                   @@\
    PATH=$${JAVA_HOME}/bin:$${VB}/bin:$${PATH}; \                       @@\
    CLASSPATH=${CLASSES11}:GEE_JAVA11_CLASSPATH;\                       @@\
    export VBROKER_ADM; export LD_LIBRARY_PATH; \                       @@\
    export PATH; export CLASSPATH; \                                    @@\
    $${VB}/bin/idl2java modulename.idl                                  @@\
    $(TOUCH) modulename/.modulename                                     @@\
JavaPackageInternal(modulename,JAVACOMPILE)                             @@\
clean::                                                                 @@\
    $(RM) -rf modulename

/* RegisterVBServer(module,interface,classnm)
    Registers with the object activation daemon (oad)
    the indicated interface.
    module - the name following the keyword "module"
    interface - the name following the keyword interface
    classnm - the name of the java class
        which registers the interface by calling
        boa.obj_is_ready(impl), where impl is a
        java object of class interfaceImpl
        It is assumed that this class is in the default
        package.
    example:
    RegisterVBServer(Genotype,UserFactory,UserFactoryImpl)
*/
#define RegisterVBServer(module,interface,classnm)                      @@\
MACROINVOKED = _RegisterVBServer(module,interface,classnm)              @@\
install.time:: classnm.class                                            @@\
    -GEE_VISIBROKER/bin/oadutil unreg \                                 @@\
        -i module/interface -o interface \                              @@\
        -host GEE_VISIBROKER_HOST                                       @@\
    GEE_VISIBROKER/bin/oadutil reg \                                    @@\
        -i module/interface -o interface \                              @@\
        -cpp ${GEEBINDIR}/gj -a classnm \                                   @@\
        -host GEE_VISIBROKER_HOST

#define InstallFile(file, flags, dest)                                       @@\
InstallFiles(file, flags, dest)

#define InstallFileToFile(file, mode, destfile)                              @@\
install.time:: file                                                          @@\
    $(INSTALL) mode file destfile

#define ForceInstallFiles(files, mode, dest)                                 @@\
install.time::                                                               @@\
    @(NOSHERRORS; \                                                          @@\
    for i in files; do \                                                     @@\
        (SHVERBOSE; $(INSTALL) mode $$i dest/$$i); done)


/* There must be NO initial spaces on the third argument */
#define InstallFileAsNeeded(file, mode, dest)                                @@\
MACROINVOKED = _InstallFileAsNeeded(file, mode, dest)                        @@\
install::dest                                                                @@\
dest: file                                                                   @@\
    ${INSTALL} mode file dest

/* There must be NO initial spaces on the third argument */
#define SystemInstallFileAsNeeded(file, mode, dest)                          @@\
MACROINVOKED = _SystemInstallFileAsNeeded(file, mode, dest)                  @@\
systeminstall:: dest/file                                                    @@\
dest/file: file                                                              @@\
    ${INSTALL} mode file dest                                                @@\
    (df=dest/file; \                                                         @@\
    cmd=`${LSLD} dest | ${AWK} '/^d/\                                        @@\
        {print "${CHOWN} " $$3 " '"$$df"';";\                                @@\
         print "${CHGRP} " $$4 " '"$$df"'"}'`;\                              @@\
    ${SH} -c "$$cmd")

#define InstallIDLPackage(package,dir)                                       @@\
MACROINVOKED = _InstallIDLPackage(package, dir)                              @@\
MkdirTarget(dir/package)                                                     @@\
InstallFiles(package/ALLFILES.class, $(INSTCLASSFLAGS), dir)                 @@\
InstallFiles(package.idl, $(INSTDATAFLAGS), ${GEELIBDIR}/idl)
```

```
/* ****************************************************************** *\
 * MISCELLANEOUS RULES
\* ****************************************************************** */

#define AppendFiles(target, sources)                                    @@\
target: sources                                                         @@\
    $(RM) ,target target                                                @@\
    $(CAT) sources > ,target                                            @@\
    $(MV) ,target target


#define     CleanTarget(files)                                          @@\
MACROINVOKED = _CleanTarget(files)                                      @@\
clean:: ; $(RM) files

#define MkdirTarget(dirs)                                               @@\
MACROINVOKED = _MkdirTarget(dirs)                                       @@\
install.time:: makedirs                                                 @@\
makedirs::                                                              @@\
    @$(ECHO) "Checking Destination Directories...."                     @@\
    @sh -c 'for i in dirs;  do \                                        @@\
        if [ -f $$i ]; then  \                                          @@\
            $(ECHO) Mkdir: $$i is a FILE; \                             @@\
            exit 1; \                                                   @@\
        elif [ ! -d $$i ]; then  \                                      @@\
            $(ECHO) Making directory $$i; \                             @@\
            $(CONFIGDIR)/geemkdirs $$i;\                                @@\
        fi; \                                                           @@\
    done; \                                                             @@\
    exit 0'


/* ****************************************************************** *\
 * RULES FOR IMAKE.TMPL
\* ****************************************************************** */

#define RecursiveMakes(subdirs)                                         @@\
MACROINVOKED = _RecursiveMakes(subdirs)                                 @@\
CompileSubdirs(subdirs)                                                 @@\
InstallSubdirs(subdirs)                                                 @@\
JavalistSubdirs(subdirs)                                                @@\
GEESubdirs(subdirs)                                                     @@\
SystemInstallSubdirs(subdirs)                                          @@\
CleanSubdirs(subdirs)


#define    MakefileTarget()                                            @@\
Makefile:: Imakefile $(CONFIGDIR)/imake.tmpl                           @@\
        $(CONFIGDIR)/gee.rls $(CONFIGDIR)/gee.h                        @@\
   $(IMAKE_CMD)                                                        @@\
$(CONFIGDIR)/gee.h: $(CONFIGDIR)/gee.properties                        @@\
   $(ECHO) "*** config/gee.h is out of data. redo 'make Makefiles'"    @@\
   exit 1

#define SubdirTarget(dirs, mktag, string)                              @@\
mktag::                                                                @@\
   @(NOSHERRORS; \                                                     @@\
   for i in dirs; do \                                                 @@\
     (cd $$i; $(ECHO) string "(`$(PWD)`)"; \                           @@\
      $(MAKE) $(MFLAGS) CONFIGDIR=$(CONFIGDIR) \                       @@\
         JAVADOCLIMB=$(JAVADOCLIMB)/$$i \                              @@\
         JAVADOCTOP=../$(JAVADOCTOP) \                                 @@\
         mktag || exit 1 ) || exit 1; done)

/****************************************************************
NOTE: The following uses of SubdirTarget() have no spaces between the
arguments because the HP version of imake does some weird expansion
of the space into a tab, which causes the target tag to be tabbed
and, thus, unrecognized by make.
****************************************************************/

#define CompileSubdirs(dirs)                                           @@\
SubdirTarget(dirs,Compiles,"compiling")

#define InstallSubdirs(dirs)                                           @@\
SubdirTarget(dirs,Installs,"installing")

#define JavalistSubdirs(dirs)                                          @@\
SubdirTarget(dirs,javalist,"listing java files")

#define SystemInstallSubdirs(dirs)                                     @@\
SubdirTarget(dirs,SystemInstall,"system installation")

#define CleanSubdirs(dirs)                                             @@\
SubdirTarget(dirs,Clean,"cleaning")
```

```
/*
    'make GEE' at any level will make Makefiles in subdirs and
        then install in self and GEE in subdirs
    'make Makefiles' will make Makefile and then make subMakefiles
    'make subMakefiles' will build a Makefile in each subdirectory
        and then make subMakefiles in each subdirectory
*/

#define GEESubdirs(dirs)                                            @@\
GEE::                                                               @@\
    @NOSHERRORS; \                                                  @@\
    for i in dirs; do \                                             @@\
        $(ECHO) "Making sub Makefiles (`$(PWD)`/$$i)";\             @@\
        $(MAKE) $(MFLAGS) subdirMakefile \                          @@\
            CONFIGDIR=$(CONFIGDIR) \                                @@\
            MAKE_SUBDIR=$$i || exit 1;\                             @@\
    done                                                            @@\
                                                                    @@\
SubdirTarget(dirs,GEE,"building GEE")                               @@\
                                                                    @@\
Makefiles:: subMakefiles                                            @@\
subMakefiles::                                                      @@\
    @NOSHERRORS; \                                                  @@\
    for i in dirs; do \                                             @@\
        $(ECHO) "Making Makefiles (`$(PWD)`/$$i)"; \                @@\
        $(MAKE) $(MFLAGS) subdirMakefile \                          @@\
            CONFIGDIR=$(CONFIGDIR) \                                @@\
            MAKE_SUBDIR=$$i || exit 1;\                             @@\
        (cd $$i; $(MAKE) $(MFLAGS) \                                @@\
            CONFIGDIR=$(CONFIGDIR) \                                @@\
                subMakefiles || exit 3); \                          @@\
    done                                                            @@\
subdirMakefile:                                                     @@\
    cd $(MAKE_SUBDIR);  $(IMAKE_CMD)
```

Appendix D: The Imake Template: imake.tmpl

Imake essentially just processes this template file, *imake.tmpl*, through the C preprocessor, using the command line to set INCLUDE_IMAKEFILE to the pathname of the chosen *Imakefile*. That file and other definition files are incorporated by the #include statements. *Make* variable definitions, indicated with an equal sign, define symbols that can be used in macros in *gee.rls* or in the *Imakefiles* themselves.

```
#ifndef XCOMM
#ifdef GNU_CPP_ENV
#define XCOMM \#
#else
#define XCOMM #
#endif
#endif
XCOMM $Id: imake.tmpl,v 1.28 1999/02/11 18:26:35 wjh Exp $
XCOMM=================================================
XCOMM Copyright Carnegie Mellon Univ. 1998 - All Rights Reserved
XCOMM=================================================

XCOMM $Disclaimer:  $

XCOMM ##################################################
XCOMM This Makefile is automatically generated by
XCOMM imake. Do not modify it or you will lose your
XCOMM changes when imake generates makefiles again.
XCOMM Ignore this message if you are not using imake.
XCOMM ##################################################

XCOMM CONFIGDIR is set by make Makefiles to ./config/ in the source tree
XCOMM and is set by genmake to ./config in the destination tree.
XCOMM
XCOMM CONFIGDIRDEFN is #defined on the command line to imake
XCOMM    and = defined on a call from 'make Makefiles'
    CONFIGDIR = CONFIGDIRDEFN
#undef CONFIGDIRDEFN
```

```
XCOMM These JAVADOC variables are for 'make Javadocs'
XCOMM They are overridden by the Make rucursion.
JAVADOCLIMB = .
JAVADOCTOP = .

#include <imakeconcat.h>

.NO_PARALLEL:

.SUFFIXES:
XCOMM suffix rules are too unpredictable. Use % macros instead.
XCOMM .SUFFIXES: .class .java .h .properties .csh .sh .idl

all::

XCOMM ############ Reading gee.h ###################
#include <gee.h>
XCOMM ############ Done reading gee.h ##############

XCOMM Destination directories
GEEBINDIR = GEE_DESTINATION_DIRECTORY/bin
GEELIBDIR = GEE_DESTINATION_DIRECTORY/lib
GEEDOCDIR = GEE_DESTINATION_DIRECTORY/doc
GEECLASSESDIR = GEE_DESTINATION_DIRECTORY/classes
GEEHTMLDIR = GEE_DESTINATION_DIRECTORY/html
GEEETCDIR = GEE_DESTINATION_DIRECTORY/etc

XCOMM Commands used in Makefiles

XCOMM JAVA 1.2
XCOMM usage of variables:
XCOMM    EXTRACLASSES - add to CLASSPATH, can be set in Imakefile
XCOMM        set to ../xxx, colon separated list
XCOMM    SITE_CLASSES - dir:dir..., can be set in site.mcr
XCOMM    SITE_JAVAC_OPTIONS, can be set in site.mcr
XCOMM    JAVAC_OPTIONS, can be set in Imakefile
XCOMM    SITE_JAVA_OPTIONS, can be set in site.mcr
XCOMM    JAVA_OPTIONS, can be set in Imakefile
XCOMM    SITE_JAVA_DEFINES - -Dxx=xx ..., can be set in site.mcr
XCOMM    JAVA_DEFINES - -Dxx=xx ..., can be set in an Imakefile


  CLASSES = .:$(EXTRACLASSES):${SITE_CLASSES}
  JCOPTIONS = ${SITE_JAVAC_OPTIONS} ${JAVAC_OPTIONS}
  JAVACOMPILE = CLASSPATH=${CLASSES} ${CONFIGDIR}/gjc \
       $(JCOPTIONS)

  OPTIONS = ${SITE_JAVA_OPTIONS} ${JAVA_OPTIONS}
  DEFINES = $(SITE_JAVA_DEFINES) $(JAVA_DEFINES) \
       -Dgee.use.sandbox=$(CONFIGDIR)/..
  JAVARUN = CLASSPATH=${CLASSES} ${CONFIGDIR}/gj \
       ${OPTIONS} ${DEFINES}

  JAVADOC = GEE_JAVA_HOME/bin/javadoc \
       -classpath
$(CLASSES):GEE_JAVA_BOOTCLASSPATH:GEE_JAVA_CLASSPATH

XCOMM JAVA 1.1

  CLASSES11 = .:$(EXTRACLASSES11):${SITE_CLASSES11}

  COPTIONS11 = ${SITE_JAVAC11_OPTIONS} ${JAVAC11_OPTIONS}
  JAVA11COMPILE = JAVA_HOME=GEE_JAVA11_HOME
CLASSPATH=${CLASSES11} \
       ${CONFIGDIR}/gjc11 $(COPTIONS11)

  OPTIONS11 = $(SITE_JAVA11_OPTIONS) $(JAVA11_OPTIONS)
  DEFINES11 = $(SITE_JAVA11_DEFINES) $(JAVA11_DEFINES) \
       -Dgee.use.sandbox=$(CONFIGDIR)/..
  JAVA11RUN = CLASSPATH=${CLASSES11} ${CONFIGDIR}/gj11 \
       ${OPTIONS11} ${DEFINES11}

  JAVADOC11 = GEE_JAVA11/bin/javadoc  -classpath $(CLASSES11)

XCOMM C and C++. Override these in site.mcr
  CC = cc
  CPP = cc -E
  CXX = g++
  IMAKE = imake

XCOMM standard UNIX utilities
  AWK = /bin/awk
  CAT = /bin/cat
  CHGRP = /bin/chgrp
  CHOWN = /bin/chown
```

```
   CO = /usr/local/bin/co
   CP = /bin/cp
   CSH = /bin/csh -f
   ECHO = echo
   INSTALL = /usr/ucb/install
   JAR = GEE_JAVA_HOME/bin/jar
   LN = /bin/ln
   LS = /bin/ls
   LSLD = /bin/ls -ld
   MKDIR = /bin/mkdir
   MV = /bin/mv
   PWD = /bin/pwd
   RM = /bin/rm -f
   RMIC = GEE_JAVA_HOME/bin/rmic
   SED = /bin/sed
   SH = /bin/sh
   TAR = /bin/tar
   TOUCH = /bin/touch
   ZCAT = /bin/zcat
   IMAKE_CMD =  $(IMAKE) -Timake.tmpl -I. -I$(CONFIGDIR) \
            -DCONFIGDIRDEFN=$(CONFIGDIR)

XCOMM modes for $(INSTALL)
   INSTMODEFLAGS = -c -m
   INSTPROGFLAGS = $(INSTMODEFLAGS) 0555
   INSTUIDFLAGS = $(INSTMODEFLAGS) 4555
   INSTCLASSFLAGS = $(INSTMODEFLAGS) 0444
   INSTDATAFLAGS = $(INSTMODEFLAGS) 0444
   INSTLIBFLAGS = $(INSTMODEFLAGS) 0555
   INSTDOCFLAGS = $(INSTMODEFLAGS) 0444
   INSTMANFLAGS = $(INSTMODEFLAGS) 0444

XCOMM ################ Reading gee.rls #########################
#include "gee.rls"
XCOMM ################ Done reading gee.rls ####################

XCOMM ################ Reading site.mcr #######################
#include "site.mcr"
XCOMM ################ Done reading site.mcr ##################

#if ConstructMFLAGS
      MFLAGS = -$(MAKEFLAGS)
#endif
```

```
all:: compile
Compiles:: compile
compile::
Installs:: install
install:: compile install.time
GEE::  install systeminstall
SystemInstall:: systeminstall
Clean:: clean
Makefiles:: Makefile
subMakefiles::
systeminstall::

Javadocs javadoc:: javadoc/,javafiles javadoc/index.html

javadoc/,javafiles::
    $(RM) -rf javadoc
    -${MKDIR} javadoc
    $(TOUCH) javadoc/,javafiles
javadoc/,javafiles:: javalist
javalist::
javadoc/index.html:: javadoc/,javafiles
    $(JAVADOC) -d javadoc `sort -n javadoc/,javafiles`
    $(RM) javadoc/,javafiles

XCOMM ####################################################
XCOMM The following comes from the local Imakefile
XCOMM ####################################################
#include INCLUDE_IMAKEFILE
XCOMM ####################################################
XCOMM Back from the local Imakefile
XCOMM ####################################################

clean::
    $(RM) -rf javadoc

#ifndef TopLevelMakefile
MakefileTarget()
#endif

%.class: %.java
    ${JAVACOMPILE} $<
```

```
%.java: RCS/%.java,v
    ${CO} $@
%.sh:  RCS/%.sh,v
    ${CO}  $@
%.csh:  RCS/%.csh,v
    ${CO}  $@
%.sql:  RCS/%.sql,v
    ${CO}  $@
%.idl:  RCS/%.idl,v
    ${CO}  $@
%tar.Z: RCS/%tar.Z
    ${CP} $< .
%tar: %tar.Z
    ${ZCAT} $< > $@

clean::
    $(RM) "#"* ,* *~ *.CKP *.BAK *.bas errs core *.stubbed
    $(RM) *.ln *.o *.class make.log install.time foo*

install.time::
    $(TOUCH) install.time

listdirs:
    @$(ECHO) ${SUBDIRS}
```

Appendix E: Top-Level Imakefile

In the top-level directory of the source tree, special processing is necessary to get the initial *Makefile*. The *Imakefile* below suffices because it is constructed so that it can serve as input to *xmkmf* or *make* and will in either case result in building a *Makefile* using the *Amakefile* instead of an *Imakefile*. (See Appendix F for the *Amakefile*.)

In the text below, an at-sign, @, at the beginning of a command indicates that that line is not echoed to the output. A dash preceding a command indicates that errors arising therefrom should be ignored. (For instance, deleting an absent file would otherwise terminate the *make*.) Note that the only important effect of this file is the *imake* command on the sixth line. The bulk of the file constructs *imakeconcat.h*, which defines three macros—Concat, Concat3, and Stringize. These macros are useful in *Imakefiles,* although only one appears in *gee.rls* (in RMIStubs—see Appendix C).

```
.NO_PARALLEL:

all:: config/gee.h config/gee.sed Amakefile config/imake.tmpl \
        config/site.mcr config/gee.rls config/geemkdirs \
        config/imakeconcat.h
    -@rm -f Makefile
    imake -I. -I./config -Timake.tmpl  -f Amakefile \
            -DCONFIGDIRDEFN=`pwd`/config
config/geemkdirs:
    cc -o config/geemkdirs config/geemkdirs.c

config/gee.h config/gee.sed: config/gee.properties config/Defines.java
    javac config/Defines.java
    java -classpath config:$$CLASSPATH Defines config/gee.properties


config/imakeconcat.h:
    @rm -f Makefile testfile
    @echo '#define Concat(x,y)x##y'            > testfile
    @echo 'Concat(use,hashes)'                 >> testfile
    @imake -Ttestfile -f testfile
    @if grep usehashes Makefile ; then \
        echo "#define Concat(a,b)a##b"         > cat; \
        echo "#define Concat3(a,b,c)a##b##c"      >> cat; \
        echo "#define Stringize(a) #a"            >> cat; \
    else \
        echo "#define Concat(a,b)a/**/b"          >  cat; \
        echo "#define Concat3(a,b,c)a/**/b/**/c"  >> cat; \
        echo '#define Stringize(a) "a"'           >> cat; \
    fi
    @rm -f testfile Makefile config/imakeconcat.h
    mv cat config/imakeconcat.h

Makefile GEE Compiles Installs Javadocs Clean::
    @echo
    @echo USE  'make' WITHOUT ANY TARGET TO MAKE 'Makefile'
    @echo
Makefile::
    @echo Ignore the following message:
Makefile:: "Makefile",yet
```

Appendix F: Top-Level Amakefile

Amakefile, below, is in the top-level source directory where it functions as the *Imakefile* for the *imake* invoked by the file that is called "*Imakefile*" in that same directory. Most of the processing suggested here is actually done by the *Imakefile* (Appendix E). The only effective part is the call "RecursiveMakes($(SUBDIRS))," which builds all subdirectories.

```
/* ************************************************************************
Copyright Carnegie Mellon University 1998 - All Rights Reserved
************************************************************************ */
/* $Disclaimer:  $ */


#define TopLevelMakefile

DIRS = GEE_DESTINATION_DIRECTORY  \
    GEE_DESTINATION_DIRECTORY/config  \
    ${GEEBINDIR} ${GEELIBDIR} ${GEECLASSESDIR} \
    ${GEEETCDIR} ${GEEDOCDIR} ${GEEHTMLDIR} \
    ${GEELIBDIR}/idl

MkdirTarget($(DIRS))

all:: Makefile
    $(ECHO) A new Makefile has been created. Please restart.
    exit 1

$(CONFIGDIR)/gee.h $(CONFIGDIR)/gee.sed: \
        $(CONFIGDIR)/gee.properties $(CONFIGDIR)/Defines.java
    javac $(CONFIGDIR)/Defines.java
    java -classpath $(CONFIGDIR):$$CLASSPATH Defines \
        $(CONFIGDIR)/gee.properties


Makefile:: Amakefile  $(CONFIGDIR)/imake.tmpl  $(CONFIGDIR)/gee.h \
        $(CONFIGDIR)/gee.rls  $(CONFIGDIR)/site.mcr \
        $(CONFIGDIR)/gee.h
    $(IMAKE) -f Amakefile -Timake.tmpl -I. -I$(CONFIGDIR) \
        -DCONFIGDIRDEFN=$(CONFIGDIR)
```

```
GEE::
    @$(ECHO)
_________________________________________________________________
    @$(ECHO) Copyright 1998 Carnegie Mellon University
    @$(ECHO) -
    @$(ECHO) Building GEE begins at `date`
    /* the actual build happens from the rule for "GEE"
        constructed by RecursiveMakes() */

SUBDIRS = config gee rdbms security naming busLog appclient

RecursiveMakes($(SUBDIRS))

GEE Javadocs:: install.time javadoc/index.html
    $(RM) -rf $(GEEHTMLDIR)/javadoc
    $(TAR) cf - ./javadoc | (cd $(GEEHTMLDIR); tar xf -)

GEE::
    @$(ECHO)
    @$(ECHO) Here endeth the build of GEE for `date`
    @$(ECHO) =
    @$(ECHO) To install documentation and html files:
    @$(ECHO) "   cd ../doc; rehash; genmake; make GEE"
    @$(ECHO) =
    @$(ECHO)
_________________________________________________________________
```

Appendix G: genmake.csh

In an installed system, *genmake* generates a *Makefile* from an *Imakefile* using the configuration files installed in the *config* directory of the installed system. *Genmake* is installed by processing *genmake.csh*, below, through *sed* with *gee.sed* as the *sed* command. This replaces the string GEE_DESTINATION_DIRECTORY with its value from *gee.properties*.

```
#!/bin/csh -f
# directory TARGET/config has the GEE configuration files
# TARGET is set by the first defined of
#   a) first argument on command line
#   b) environment variable GEEDIR
#   c) build time value for GEE_DESTINATION_DIRECTORY
# subsequent entries on the command line are taken as arguments to imake
# to give arguments without specifying the TARGET, use --- as the first arg

if ($#argv>0 && "$1" != "---") then
    set TARGET = $1
    shift
else if (${?GEEDIR}) then
    set TARGET = ${GEEDIR}
else
    set TARGET = GEE_DESTINATION_DIRECTORY
endif
if ("$1" == "---") then
    shift
endif
if (! -e Imakefile) then
    echo "*** No Imakefile - Cannot Generate Makefile"
    exit(1)
endif

echo "Generating Makefile using defns in ${TARGET}/config"
mv Makefile ,Makefile >& /dev/null
imake -T imake.tmpl -I${TARGET} -I${TARGET}/config \
        -DCONFIGDIRDEFN=${TARGET}/config $argv:q
rm -f ,Makefile
```

Appendix H: needjc.c

The GEE script for Java compilation uses *needjc*, below, to generate the list of source files for a *javac* compilation.

```
/* needjc.c
    Prints those of its arguments which are the names of existing java files
    for which there is no class file or the java file is newer than the class file.
*/
#include <sys/types.h>
#include <sys/stat.h>

static long filetime(fnm)  char *fnm;  {
    struct stat buf;
    if (stat(fnm, &buf) != 0) return 0L;
    return buf.st_ctime;
}
static int needscompiled(jnm)   char *jnm; {
    char cnm[5000];
    int dotloc;
    long ctm, jtm;
    strcpy(cnm, jnm);
    dotloc = strrchr(cnm, '.');
    if (dotloc ==0 || strcmp(dotloc, ".java") != 0)  return 0;  /* not a .java file */
    strcpy(dotloc, ".class");
    jtm = filetime(jnm);
    if (jtm == 0) return 0;
    ctm = filetime(cnm);
    return ctm == 0 || jtm > ctm;
}
main(argc, argv)   int argc;  char **argv;  {
    int i;
    for (i = 1; i < argc; i++) {
        if (needscompiled(argv[i]))
            printf("%s\n", argv[i]);
    }
}
```

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY (LEAVE BLANK)	2. REPORT DATE January 2000	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Construction and Deployment Scripts for COTS-Based, Open Source Systems		5. FUNDING NUMBERS C — F19628-95-C-0003
6. AUTHOR(S) Wilfred J. Hansen		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-99-TR-013
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-99-013
11. SUPPLEMENTARY NOTES		
12.A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12.B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

Construction/deployment scripts direct the compilation of sources to executables and the installation of those executables. This report details the construction/deployment scripts developed at the Software Engineering Institute for the GEE project. GEE, a Generic Enterprise Ensemble, is a prototypical three-tier information system incorporating a number of commercial off-the-shelf (COTS) products. The scripts for GEE were challenging because we wanted a self-contained package of scripts and source files from which the system could be built and deployed either by us at our site or by customers at their sites. The COTS products we used—Java, an Oracle database, the Visibroker implementation of CORBA (the Common Object Request Broker Architecture), and the Netscape Web browser and server—added challenges in installation, version change, process initiation, and communication rendezvous. This report describes the challenges and how our solutions exploited the principles of "Repeat not" and "Delay binding." Lessons learned are reported elsewhere.

SUBJECT TERMS), **commercial off-the-shelf (COTS) products, Common Object Request Broker Architecture (CORBA), construction/deployment scripts, Generic Enterprise Ensemble (GEE)**			15. NUMBER OF PAGES 86
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102